Clube Delphi

Relatórios com FastReport
Construa relatórios de forma simples

ListView em apps mobile
Aprenda a personalizar
esse componente
em tempo de execução

Edição 170 :: R$ 14,90

DEVMEDIA

MULTICAMADAS E REST

Desenvolva e consuma serviços DataSnap

ISSN 1517990-7

MVP

R$ 1.000.000,00
INVESTIDOS EM CONTEÚDO NOS ÚLTIMOS 12 MESES.

APLIQUE ESSE INVESTIMENTO NA SUA CARREIRA...

E MOSTRE AO MERCADO QUANTO VOCÊ VALE!

CONFIRA TODO O MATERIAL QUE VOCÊ TERÁ ACESSO:

- + de **9.000** video-aulas
- + de **290** cursos online
- + de **13.000** artigos
- DEVMEDIA API's consumido + de **500.000** vezes

POR APENAS
R$ 69,90* mensais

*Tempo mínimo de assinatura: 12 meses.

ASSINATURA MVP MOST VALUABLE PROFESSIONAL

PRA QUEM QUER EXIGIR
MAIS DO MERCADO!

DEVMEDIA

Clube Delphi

170ª Edição 2016 ISSN 1517990-7 Impresso no Brasil

Editor Geral

Joel Rodrigues (joelrlneto@gmail.com)

Jornalista Responsável

Kaline Dolabella - JP24185

Revisora Técnica

Anny Caroline (annycarolinegnr@gmail.com)

Capa e Diagramação

Romulo Araujo

Atendimento ao Leitor

A DevMedia conta com um departamento exclusivo para o atendimento ao leitor. Se você tiver algum problema no recebimento do seu exemplar ou precisar de algum esclarecimento sobre assinaturas, exemplares anteriores, endereço de bancas de jornal, entre outros, entre em contato com:

www.devmedia.com.br/central
(21) 3382-5038

Publicidade

Para informações sobre veiculação de anúncio na revista ou no site e para fechar parcerias ou ações específicas de marketing com a DevMedia, entre em contato com:

publicidade@devmedia.com.br

Distribuição

FC Comercial e Distribuidora S.A
Rua Teodoro da Silva, 907
Grajaú - RJ - 206563-900

MVP

Assine agora e tenha acesso a todo o conteúdo da DevMedia: **www.devmedia.com.br/mvp**

DEVMEDIA

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique a vontade para entrar em contato com os editores e dar a sua sugestão!

Se você estiver interessado em publicar um artigo na revista ou no site ClubeDelphi, entre em contato com os editores, informando o título e mini-resumo do tema que você gostaria de publicar:

Joel Rodrigues - Editor da Revista
joelrlneto@gmail.com

Joel Rodrigues - **Editor Geral**

Editor da ClubeDelphi Magazine

Sumário

04 – Como criar relatórios no Delphi com FastReport
[*Gutierry Antonio*]

Artigo no estilo solução completa

16 – Multicamadas e REST com DataSnap
[*Jones Granatyr*]

Destaque - Vanguarda

25 – Personalização do ListView em aplicativos mobile
[*Jones Granatyr, Aldo Marcelo Paim, Fabio Spak, Graziele Katlen e Fábio Alexandre Taffe*]

Como criar relatórios no Delphi com FastReport

Aprenda a construir relatórios com qualidade e produtividade

Relatórios são ferramentas amplamente utilizadas em diversas áreas e situações para apresentar, por exemplo, detalhes sobre um experimento ou evento, bem como para expor dados em um determinado contexto. A composição e uso desses relatórios costuma variar bastante, tanto entre sistemas, quanto dentro de um mesmo software. Por exemplo, em um sistema de gerenciamento de eventos, possivelmente será necessária uma listagem dos convidados/inscritos, normalmente utilizada para controlar a entrada na portaria. Já para uma empresa que trabalha com vendas, pode ser interessante analisar quais são os produtos mais vendidos, ou mesmo obter uma lista dos clientes que se encontram com algum débito no momento, a fim de auxiliar na tomada de decisões estratégicas. Nesses casos, além de simples listagens, também é comum que se encontre agrupamentos, somatórios e gráficos.

Tendo em mente a importância dos relatórios, e considerando que em Delphi é comum o desenvolvimento de aplicações de automação comercial e gerenciamento empresarial (ERPs), conheceremos nesse artigo uma das ferramentas de relatório mais utilizadas na atualidade: o FastReport.

Por que usar o FastReport?

Para compreendermos bem porque utilizar o FastReport, é importante conhecer um pouco mais sobre essa ferramenta, que foi desenvolvida pela Fast Reports Inc., fundada em 1998, com o intuito de apoiar a criação de relatórios, originalmente para o Delphi e Kylix. Desde então, ele vem recebendo constantes melhorias e superando concorrentes como o QuickReport, o ReportBuilder, o RaveReports e o Crystal Reports. Em 2003, foi lançada uma versão com recursos de exportação para BMP, JPEG e TIFF, e, ainda nesse ano, a empresa inovou com o lançamento da biblioteca FastScript, que permite a aplicação de scripts Pascal, C++, Basic e JavaScript no desenvolvimento de relatórios, expandindo a criação dos mesmos e dando muito mais liberdade no desenvolvimento e tratamento de conteúdo.

Fique por Dentro

Neste artigo será apresentada a ferramenta FastReport, que permite a criação e customização de relatórios tanto durante o desenvolvimento do projeto no Delphi, quanto pelos usuários finais, mesmo que eles não possuam conhecimentos de programação. Serão mostrados exemplos que vão desde a criação de relatórios simples, nos quais só são exibidos os dados de uma tabela, até recursos mais avançados, como o uso de system variables, relatórios mestre-detalhe e exportação.

Em 2004, foi liberado o FastReport3 VCL, que possibilitava a criação de um servidor de relatórios. No ano seguinte, a empresa lançou o FastReport Studio, tecnologia que permitia que usuários finais, sem qualquer conhecimento de programação, pudessem criar relatórios apenas partindo do conhecimento da base de dados e seus campos (que, nesse caso, precisava ser fornecido pelo desenvolvedor do sistema mediante alguma política de segurança).

Em 2008, seu núcleo foi alterado e a versão FastReport4 VCL foi liberada. Nesse ano, a ferramenta já contava com um melhor design e suportava o desenvolvimento de soluções OLAP (*Online Analytical Processing*). Em 2009, surgiu um novo produto, o FastReport.Net, uma versão escrita inteiramente em C#, com suporte a MVC e de uso exclusivo do Microsoft Visual Studio.

Hoje, o FastReport está disponível para as plataformas VCL e FMX da Embarcadero (Delphi e C++ Builder), Lazarus, .NET e Mono, e se tornou tão importante que foi incorporado por padrão nas novas versões do Delphi.

Essa ferramenta permite implementações simples, reuso de código, uso de eventos e faz acesso direto ao banco de dados. Uma outra grande vantagem do seu uso é a capacidade de distribuir os relatórios de forma isolada da aplicação, não tendo que gerar um build para o envio de um simples relatório.

Nota

A versão distribuída juntamente com o Delphi é uma versão mais simples, na qual não estão disponíveis alguns recursos, tais como FastCube, FastScript, entre outros. Nesse artigo foi utilizado o FastReport 5.0 Enterprise, juntamente com o Delphi Seattle.

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Componentes do FastReport

Serão mostradas aqui as duas principais paletas do FastReport: a FastReport e a FastReport Exports. Na primeira paleta estão presentes os principais componentes de criação e manipulação do FastReport (veja a **Tabela 1**), e, na segunda, os componentes específicos para cada tipo de exportação disponível (veja a **Tabela 2**).

Na paleta FastReport Exports estão disponíveis diversas opções de exportação. Isso facilita muito na hora de exportar um relatório para algum padrão, pois evita que tempo seja gasto buscando uma solução específica ou desenvolvendo algoritmos para fazer esse trabalho, como ocorre em alguns geradores de relatórios disponíveis no mercado.

Ícone	Nome	Descrição
	TfrxReport	Abre o editor de relatórios e permite que o relatório seja construído em tempo de design.
	TfrxUserDataSet	Incorpora dados, como arrays ou fontes de dados de usuários.
	TfrxDesigner	Permite manipular o design dos relatórios.
	TfrxPreview	Permite que o desenvolvedor crie seu próprio preview.
	TfrxDBDataSet	Permite incorporar datasets do projeto ao relatório.
	TfrxChartObject	Permite incorporar gráficos ao relatório.

Tabela 1. Componentes da paleta FastReport

Conhecendo o editor do FastReport

O editor do FastReport é uma janela semelhante à apresentada na **Figura 1**. Através dele será desenvolvido não só o design dos relatórios, mas também codificações e acessos a bancos de dados.

Ele é dividido em áreas de design, paletas de componentes, barras de ferramentas e painéis.

Ícone	Nome	Descrição
	TfrxXLSExport	Faz a exportação do relatório para uma planilha Excel.
	TfrxXMLExport	Faz a exportação do relatório para um arquivo XML.
	TfrxMailExport	Faz o envio de um relatório para um e-mail.
	TfrxHTML5DivExport	Faz a exportação do relatório para um arquivo HTML.
	TfrxPDFExport	Faz a exportação do relatório para um arquivo PDF.
	TfrxJPEGExport	Faz a exportação do relatório para uma imagem JPEG.
	TfrxCSVExport	Faz a exportação do relatório para um arquivo CSV.

Tabela 2. Componentes da paleta FastResport XX Exports

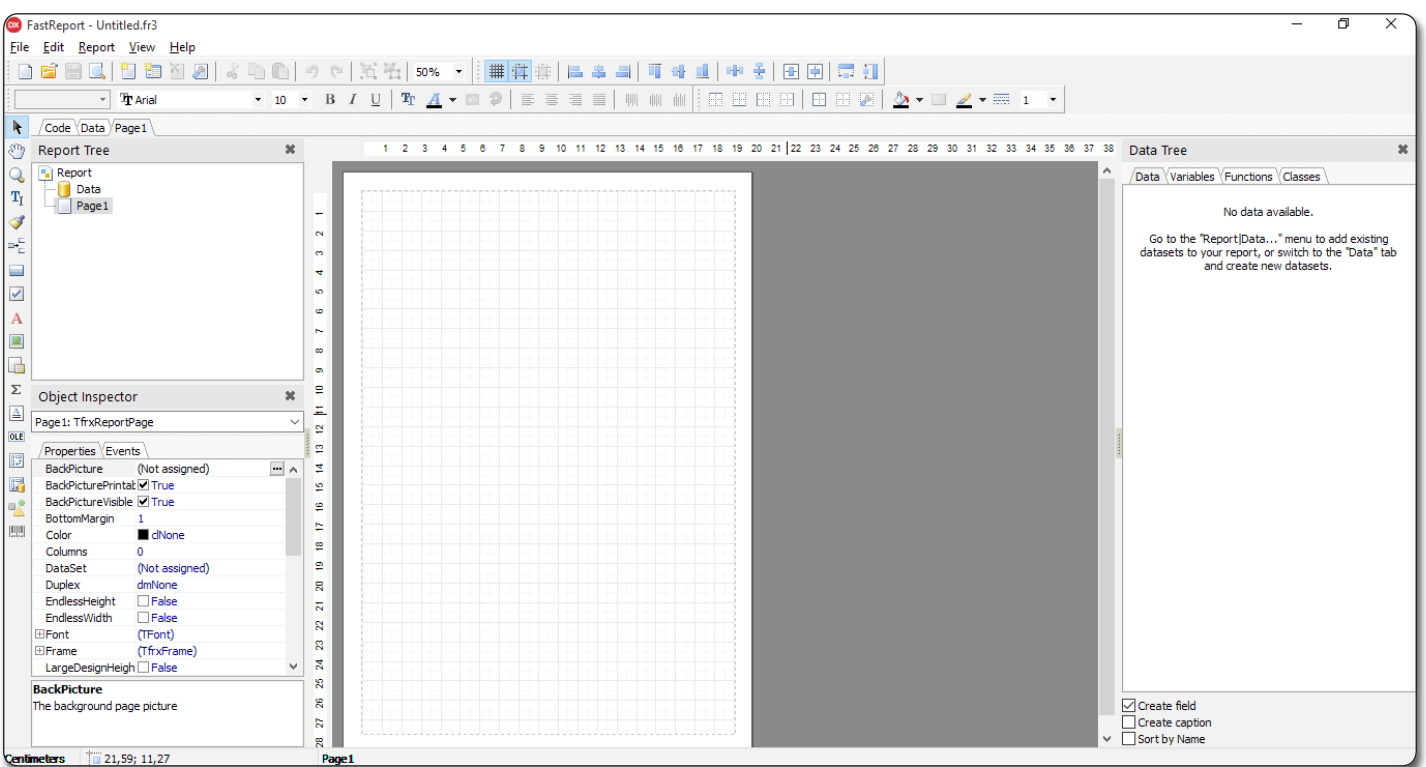

Figura 1. Editor do FastReport

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

A seguir, apresentaremos esses recursos, os quais podem ter seus locais de apresentação customizados ou serem apresentados/escondidos através do menu *View* > *ToolBars*:

1 - Standard: A barra de ferramentas *Standard* fornece botões para preview, zoom, criar e abrir relatórios, entre outros recursos (veja a **Figura 2**);

2 - Text: A *Text* disponibiliza recursos para diagramação de texto, como mostrado na **Figura 3**;

3 - Frame: Em *Frame*, temos uma paleta para alterar o design da página, na qual encontramos recursos como contorno e cor de fundo (**Figura 4**);

4 - Alignment Palette: A paleta *Alignment Palette* disponibiliza recursos de alinhamento de componentes, e é extremante útil para manter a boa aparência dos relatórios (veja a **Figura 5**);

5 - Object Inspector: O *Object Inspector* do FastReport, apresentado na **Figura 6**, é semelhante ao do Delphi e como esse ele permite a alteração de propriedades de um componente ou da página, que vão desde cor de fonte até a orientação da página ou nome do componente. Do lado direito ele disponibiliza a aba *Events*, na qual são apresentados os eventos disponíveis para um determinado componente. Nessa aba podem ser adicionados tratamentos a eventos, assim como no Delphi — ao clicar duas vezes sobre um evento, o escopo de código do mesmo é criado;

6 - Data Tree: O *Data Tree* (exibido na **Figura 7**) apresenta uma árvore de dados, na qual ficam disponíveis as tabelas e colunas. Variáveis, funções e classes também podem ser acessadas, por meio de abas separadas;

7 - Report Tree: O *Report Tree* (**Figura 8**) apresenta a árvore de componentes presentes no projeto, assim como no Delphi. Se um componente qualquer for adicionado a uma página do seu relatório, a árvore será expandida apresentando esse novo item.

Figura 2. Barra de ferramentas Standard

Figura 3. Barra de ferramenta Text

Figura 4. Barra de ferramentas Frame

Figura 5. Barra de ferramentas Alignment Palete

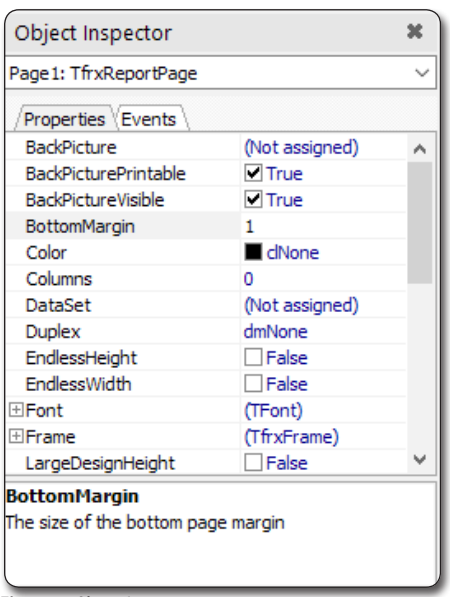

Figura 6. Object Inspector

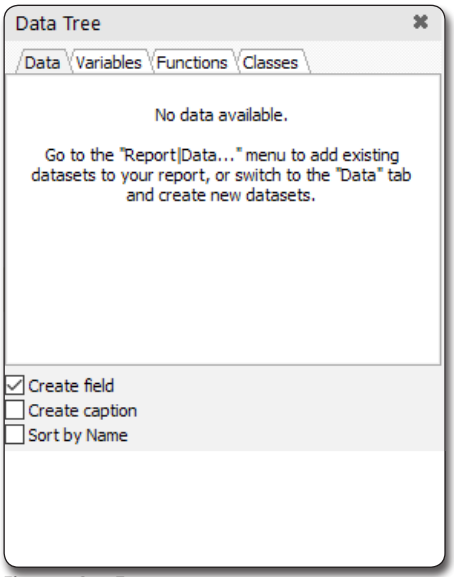

Figura 7. Data Tree

Por fim, temos as áreas de design, que são:

1 - Page designer: A *Page designer* (veja a **Figura 9**) é a área na qual são inseridos os itens a serem exibidos, tais como título, bandas, colunas e rodapés;

2 – Data: Aba em que estão disponíveis os componentes de acesso a bancos de dados, tais como connection, table e query (veja a **Figura 10**);

3 – Code: A *Code* é a área do FastReport em que possível criar scripts e interagir com a área de design, assim como se interage com os forms no Delphi através do fonte (**Figura 11**).

Modelagem do projeto

A partir de agora, desenvolveremos um projeto no qual utilizaremos o FastReport para praticar e fixar os conceitos vistos até aqui.

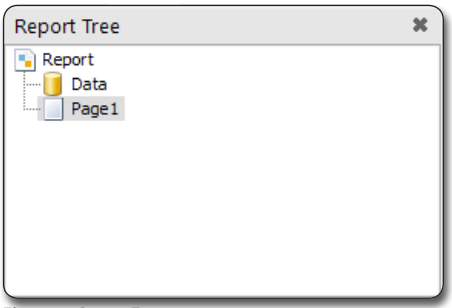

Figura 8. Report Tree

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Ícone	Nome	Descrição
	Select Tool	Permite a seleção dos componentes.
	Hand Tool	Permite que se navegue no relatório sem que haja seleção ao clicar sobre um item.
	Zoom Tool	Permite Zoom In ou Zoom Out sobre uma área.
	Insert Band	Abre um subgrupo com os tipos de bandas disponíveis.
	Text Object	Cria um objeto de texto, semelhante ao label do Delphi.
	Picture Object	Permite que imagens sejam adicionadas ao relatório.
	Subreport Object	Permite que outro relatório seja anexado ao atual.
	System Text	Cria um controle semelhante a um label, porém com predefinições de variáveis do sistema, como data, hora, número da página, entre outras.
	Rich Text Object	Adiciona um controle semelhante a um memo, indicado para observações e textos.
	Chart Object	Permite que sejam adicionados gráficos à página.
	Draw	Abre um subgrupo com formas básicas, como setas, retângulos e círculos, que podem ser desenhadas no relatório.

Tabela 3. Principais componentes disponíveis no relatório

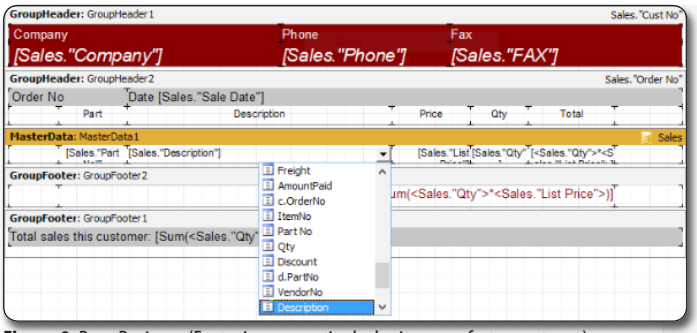

Figura 9. Page Designer (Fonte: imagem retirada do site www.fast-report.com)

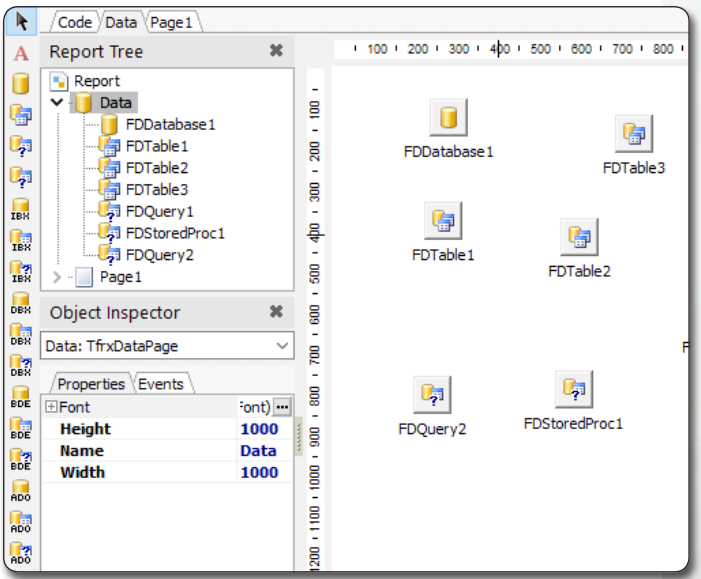

Figura 10. Aba Data

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

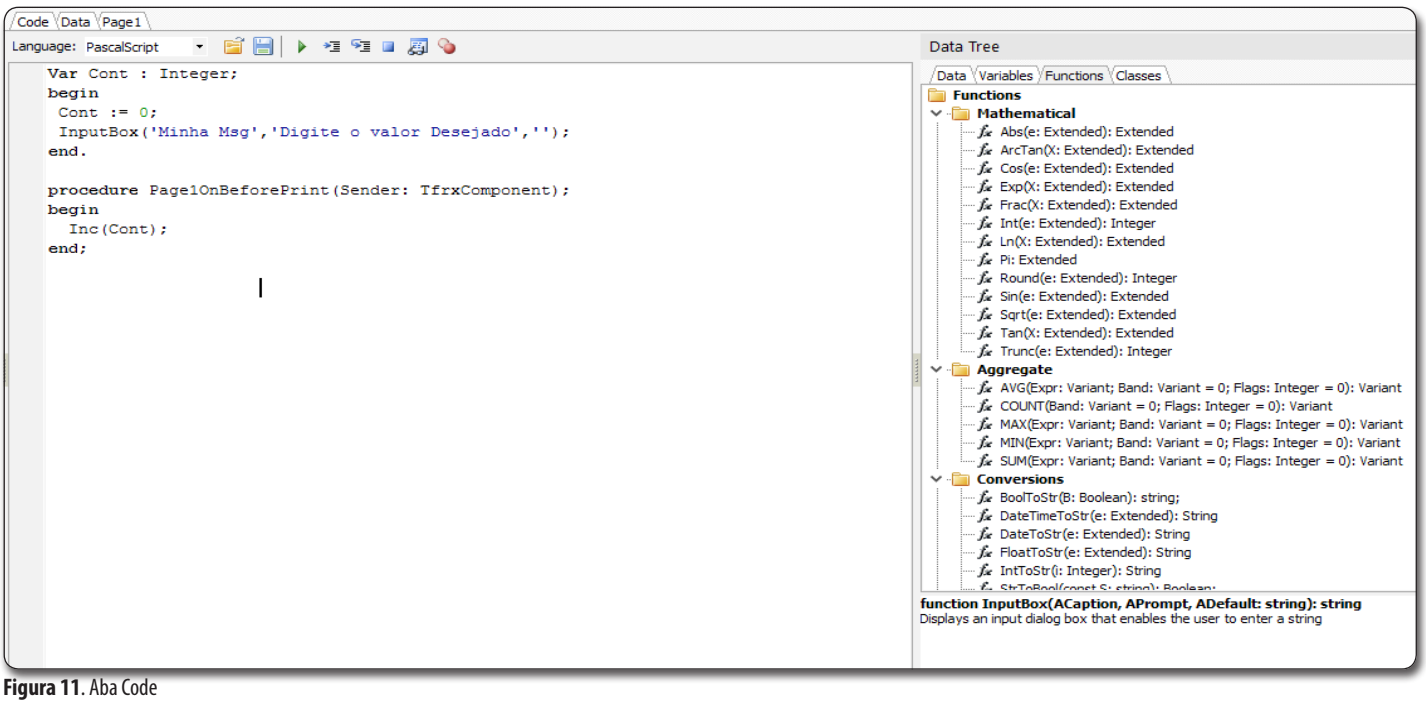

Figura 11. Aba Code

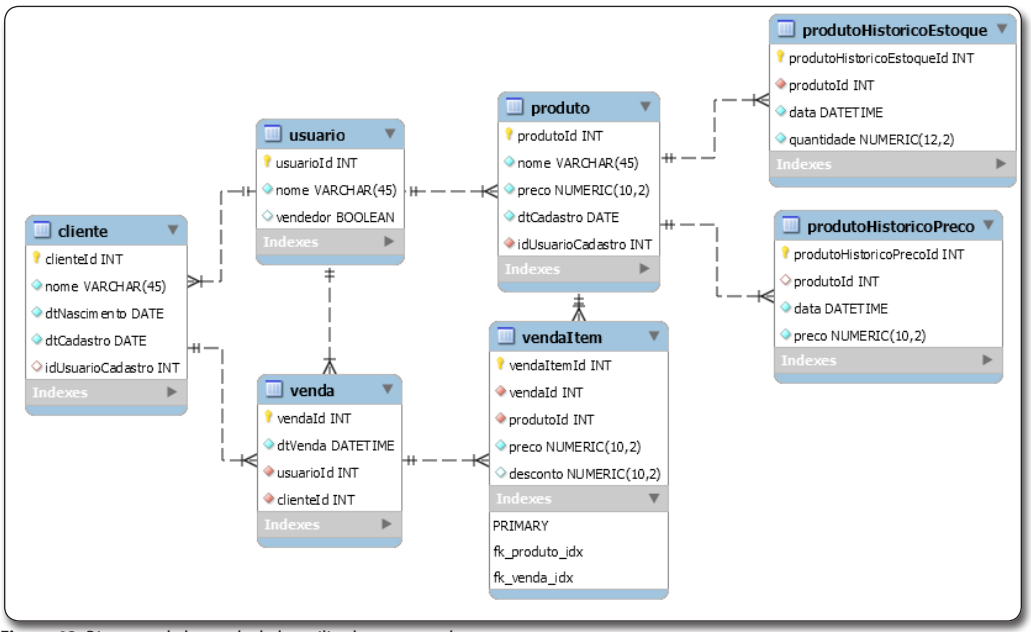

Figura 12. Diagrama do banco de dados utilizado nos exemplos

Figura 13. Apresentação da aplicação de relatórios

No entanto, antes de dar início, é necessário um projeto de dados. Para isso, será utilizada a modelagem de banco de dados mostrada na **Figura 12**, a qual será referência para a montagem dos relatórios desse artigo e a criação de um banco de dados MySQL chamado *fastreportapp*.

A partir da estrutura do banco pode ser observado que a cada alteração no escopo do produto, seja ela de preço ou de quantidade, um registro na tabela de histórico de preço ou na tabela de histórico de quantidade é gerado, permitindo remontar cenários do produto em um determinado momento na linha do tempo. Outra tabela muito importante nessa aplicação é a tabela de venda, a qual faz a ligação do cliente que fez a compra com o usuário e os itens dessa venda.

Criando um relatório simples

O primeiro relatório será bem simples, e nele serão exibidos apenas os clientes cadastrados na base de dados. Para isso, crie uma nova aplicação no Delphi, e, no formulário principal, adicione um menu e um submenu como mostrado na **Figura 13.**

Para começarmos a criar o nosso primeiro relatório, adicione um componente *TfrxReport* à aplicação e o renomeie para *frxReport_RelacaoClientes*.

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Esse componente, como explicado na **Tabela 1**, é o centro da criação de relatórios através do FastReport, e a partir dele é que serão desenvolvidos os relatórios em tempo de design. Ele também é utilizado para apresentar ou carregar um relatório para o usuário, ou para o desenvolvedor, para que ele edite relatórios já criados. Depois de adicionado o componente, clique duas vezes sobre ele para abrir o editor.

Como já estamos familiarizados com o editor, daremos início ao desenvolvimento do primeiro relatório. Na paleta de componentes presente no lado esquerdo, clique na opção *Insert Band* para que sejam apresentadas as opções de banda. Observe na **Figura 14** que o FastReport disponibiliza uma série de tipos de bandas a serem inseridas, desde Title e Summary até Subdetail, Group e bandas verticais. Adicione ao projeto uma Report Tile e um Report Summary, e, além disso, adicione uma imagem e alguns labels para que o seu projeto fique semelhante ao da **Figura 15**. Para adicionar os labels utilize o componente *Text Object*, e para alterar o tamanho das fontes e cores utilize o Object Inspector ou a barra de texto localizada no topo.

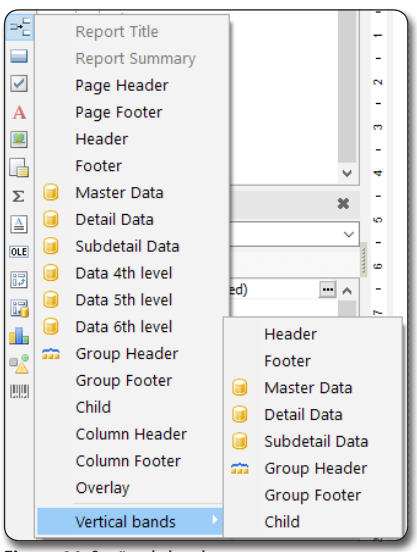

Figura 14. Opções de bandas

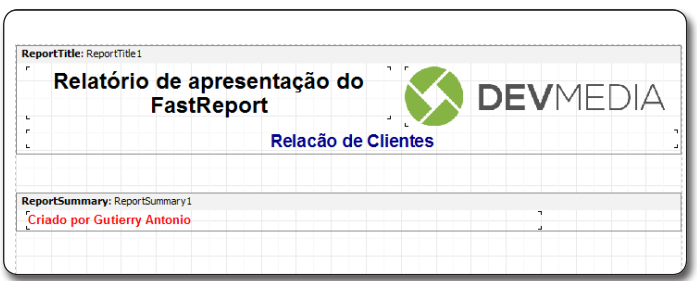

Figura 15. Montagem do relatório

Para testar o que será apresentado para o usuário, o FastReport disponibiliza previews em tempo de design, bastando utilizar o botão preview no menu ou pressionar as teclas *Ctrl + P*.

Porém, para apresentar os dados, ainda é preciso fazer a conexão com a tabela de clientes. Para configurar um acesso ao banco de dados, vá até a aba *Data*. Entre as várias maneiras disponibilizadas pelo FastReport para realizar a conexão (via FireDAC, BDE, ADO, DBX e IBX), escolheremos neste artigo a FireDAC por se tratar do mecanismo padrão das versões mais novas do Delphi.

Deve-se, agora, criar a conexão, bastando para isso adicionar à área branca do projeto o componente *FDDatabase*, renomeá-lo para *database* e, com dois cliques sobre o mesmo, abrir o *Connection Wizard*. Depois, clique sobre o botão "..." da opção *Choose the database* para que a janela de configuração de conexão do FireDAC seja apresentada. Na tela de configuração, preencha os campos com os dados de sua conexão e, depois do teste, clique em *ok*. Com a conexão definida, adicione um *FDQuery*, o renomeie para *qryCliente* e altere o valor da sua propriedade *UserName* para *tblCliente* (a propriedade *UserName* define o nome que será apresentado no Data Tree, como pode ser visto na **Figura 16**). Em seguida, clique duas vezes sobre o componente de query e adicione a seguinte consulta SQL: *SELECT * FROM CLIENTE*.

Com a conexão e o acesso à tabela definidos, retorne ao editor da *Page1*. Perceba que na aba *Data* da árvore de dados é apresentada a tabela *tblCliente* com os seus campos (veja a **Figura 16**).

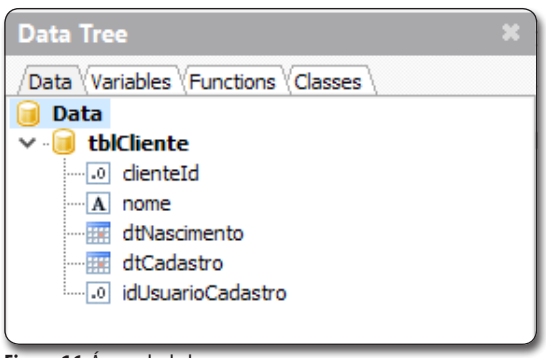

Figura 16. Árvore de dados

Em seguida, adicione à página uma banda *Master Data*. Ao escolher esse componente é necessário escolher qual dataset será responsável por controlar essa banda - nesse caso escolha a opção *tblCliente* e pressione *OK*. Nas propriedades do *Master Data* ainda é possível definir um filtro e o número de registros a serem apresentados.

Até aqui tudo está configurado corretamente, mas se solicitarmos um preview, nada será apresentado. Para que a informação seja mostrada, deve-se adicionar o conteúdo desejado dentro da banda *Master Data*. Basta arrastar as colunas desejadas da aba *Data* da *Data Tree* para o relatório. Neste exemplo serão adicionados os campos *clientId*, *nome*, *dtNascimento* e *dtCadastro*.

Para concluir esse relatório, adicione uma banda *Header* e arraste a mesma para cima da *Master Data* para que possam ser colocados os títulos de todas as colunas, resultando no relatório exibido na **Figura 17**. Caso deseje apresentar informações como a data e hora de geração do relatório, número da página atual ou o total de páginas, pode-se utilizar as *System variables*, presentes na aba *variables* do *Data Tree*.

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

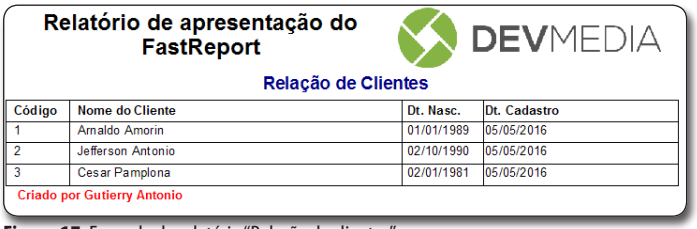

Figura 17. Exemplo do relatório "Relação de clientes"

Para concluir, salve o relatório desenvolvido e retorne ao Delphi. No evento de clique do menu "Relação de Clientes", adicione o seguinte código e execute a aplicação:

```
frxReport_RelacaoClientes.ShowReport();
```

Nota

Se ao executar a aplicação uma mensagem de erro for exibida, informando que a classe de conexão do FireDAC não foi localizada, adicione os componentes TfrxFDComponents e TFDPhysMySQLDriverLink ao formulário. O componente TfrxFDComponents faz a integração dos componentes FireDAC do Delphi com o FastReport, e o TFDPhysMySQLDriverLink é o componente de link ao driver do MySQL para acessos através do FireDAC.

Relatório em modelo de ficha

Baseado no modelo de dados definido, criaremos agora um novo relatório, em modelo de ficha, no qual será impresso um registro por página com as informações do cliente e do usuário que lhe cadastrou no sistema. Para isso, adicione um novo *TfrxReport* ao sistema e o renomeie para *frxReport_FichaDeClientes*. Depois disso, abra o editor e, na aba *Data*, adicione a conexão e uma query, cujo conteúdo será o comando SQL da **Listagem 1**.

Listagem 1. SQL da query de cliente com join no usuário

```
01 SELECT *
02 FROM CLIENTE
03 LEFT  JOIN USUARIO ON
04      (USUARIO.USUARIOID = CLIENTE.IDUSUARIOCADASTRO)
```

Aqui realizamos uma consulta simples à tabela de clientes com um join na tabela de usuários, assim identificando o usuário responsável pelo cadastro. O próximo passo é retornar à área de design e adicionar uma banda *Master*. Altere o tamanho da banda e, com o uso de linhas e dos campos da tabela de clientes e de usuários, crie uma ficha semelhante à **Figura 18**.

Ao visualizar o preview, observe que um item ficou abaixo do outro, mas a ideia é que seja exibido um cliente por página. Para isso, clique com o botão direito sobre a banda *MasterData1* e marque a opção "Iniciar em nova página" (também pode ser utilizado o *Object Inspector*, selecionando a propriedade *StartNewPage*). Com isso, ao exibir o relatório cada ficha será exibida em uma nova página

Para que o relatório fique mais elegante, adicione um header ao *MasterData1*, um título e uma logo (com o uso do *Picture Object*) ao header.

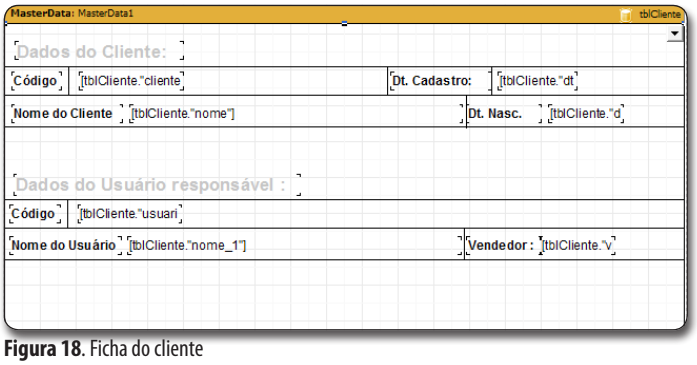

Figura 18. Ficha do cliente

Ao visualizar, perceba que apenas a primeira página exibe o header e as demais estão somente com os dados da ficha. Para que o header de uma banda seja apresentado a cada nova página, a propriedade *ReprintOnNewPage* deve ser alterada para *True*.

Se visualizarmos o relatório agora, veremos que no campo *Vendedor* é apresentada a informação "True" ou "False", mas o que desejamos é que sejam apresentadas as palavras "Sim" ou "Não". Nesse caso, poderia ser feito um *case* na consulta SQL para retornar essa informação como queremos, mas o FastReport, através do uso de expressões, permite que isso seja feito de forma fácil e elegante. Basta editar o campo *vendedor*, removendo o seu conteúdo e adicionando o seguinte comando:

```
[IIF(<tblCliente."vendedor">,'Sim','Não')]
```

Esse comando nada mais é que uma abreviação de uma estrutura de decisão If/Else, ou seja, é o equivalente ao operador ternário encontrado, por exemplo, nas linguagens C e Java. Nele, é validado se o campo *vendedor* do dataset *tblCliente* é "True": se sim, é retornado o primeiro parâmetro após a condição, e, caso contrário, é retornado o segundo parâmetro, que são, respectivamente, "Sim" e "Não" como desejamos.

Com o relatório concluído, retorne ao Delphi, crie um novo menu para a exibição do relatório e, no seu evento *onClick*, implemente o seguinte comando:

```
frxReport_FichaDeClientes.ShowReport();
```

Esse comando, como visto anteriormente, é responsável por chamar o preview do relatório em tempo de execução.

Relatório com master-detail

Agora aumentaremos a complexidade dos nossos exemplos em um novo relatório, que exibirá as vendas e seus detalhes. Portanto, crie um novo menu na tela principal, chamado "Vendas", e um submenu com o nome "Relação de vendas". Depois, adicione um novo *TfrxReport* e nomeie-o como *frxReport_RelacaoVendas*. Assim como foi feito no relatório anterior, adicione o *title* e, na aba *Data*, adicione a conexão. Depois, adicione dois componentes de query: o primeiro será o *Master* (de nome *tblVenda*), no qual deve ser adicionado o código SQL da **Listagem 2**.

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

O segundo deve ser nomeado como *tblItens*, ter sua propriedade *Master* alterada para *tblVenda* e nele devemos adicionar os comandos SQL da **Listagem 3**.

Listagem 2. SQL da query de venda

```
01 SELECT *
02 FROM VENDA
03 LEFT JOIN CLIENTE ON
04       (CLIENTE.CLIENTEID = VENDA.CLIENTEID)
```

Listagem 3. SQL da query de Itens de Venda

```
01 SELECT  *
02 FROM VENDAITEM
03 LEFT JOIN PRODUTO ON
04       (PRODUTO.PRODUTOID = VENDAITEM.PRODUTOID)
05 WHERE VENDAID = :VENDAID
```

Na **Listagem 2** temos uma consulta na tabela de vendas com um join na tabela de clientes, assim retornando os dados do pedido mais os dados do cliente. Já na **Listagem 3** é feita uma consulta na tabela de itens da venda com um join na tabela de produtos para que se tenha os dados dos produtos de uma venda. Pode ser observado que na linha 5 da **Listagem 3** existe uma condição em que o valor da coluna *vendaId* deve ser igual ao valor de um parâmetro nomeado como *vendaId*, que deve ser preenchido com o valor do id da tabela de venda. Para que isso seja possível, a propriedade *Master* deve estar devidamente preenchida, e o valor do parâmetro deve estar definido. Podemos defini-lo clicando sobre o botão "Edit query parameters", como mostrado na **Figura 19**.

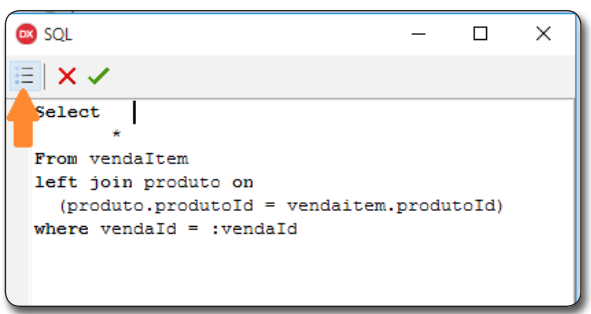

Figura 19. Botão Edit query parameters query de itens de uma venda

Em seguida, será exibida a janela *Parameters Editor*, que apresenta todos os parâmetros definidos na consulta. Nela deve ser definido o tipo de cada parâmetro e o seu valor. Para a propriedade de valor pode ser adicionado um valor fixo ou pode-se utilizar o botão "fx", que, quando clicado, apresenta a tela *Expression Editor*, semelhante ao *Data Tree*. Nessa tela podemos escolher um valor referente a um dado de um conjunto de tabela ou uma variável disponível em *variables*. Uma grande vantagem é que, além de podermos selecionar um valor de uma coluna, podemos também fazer o uso de expressões com concatenações. Para esse exemplo deve ser selecionada a coluna *vendaId* da *tblVenda*.

Com as consultas definidas, confira no lado direito, no *Data Tree*, se os dados das tabelas foram devidamente carregados, e em seguida retorne ao editor da página 1. Na page 1, além do *title*, adicione um *Master*, faça sua ligação com a query de venda e adicione na banda os campos *vendaId*, *nome* e *dtVenda*. Para adicionar um detalhe, adicione um *Detail Data*, faça a ligação do detail com a *tblItens* e na banda adicione os campos *produtoId*, *nome*, *quantidade*, *preco* e *desconto*. Adicione também um header para cada banda com os nomes das colunas apresentadas, e utilize o preview para verificar se tudo está de acordo.

Nesse momento ainda faltam informações importantes no relatório. Uma delas é o total da venda, informação essa que não está presente no modelo de dados, o qual não exibe o total geral da venda nem o total por item. Uma forma de se obter essa informação seria fazer com que o SQL retornasse o total. Calcular o total por item da venda é relativamente fácil, visto que o total é *(quantidade * preco) - desconto*. Já para retornar o total de toda a venda no SQL da query de venda, deveria ser feito um join com a tabela de itens, mas nada disso é necessário já que o FastReport auxilia esse tipo de implementação com o uso de expressões.

Para calcular o total por item de venda, adicione um *Text Object* na banda de itens da venda e dê dois cliques para editar o seu conteúdo. Observe que no topo temos o ícone "Abc" no botão *Insert Expression*. Ao clicar nele a janela *Expression Editor* é aberta, na qual podemos utilizar as colunas das tabelas para escrever a seguinte expressão:

```
(<tblItens."quantidade"> * <tblItens."preco">) - <tblItens."desconto">
```

Isso pode ser feito tanto clicando sobre as colunas desejadas quanto com a digitação livre. Após adicionar a expressão, perceba que no editor ela é apresentada dentro de colchetes ([ ]), ou seja, sempre que desejar fazer o uso de expressões, não é necessário fazer todo esse processo, basta colocá-las dentro de colchetes. Em seguida abra o preview e observe se o total está sendo apresentado corretamente.

Para adicionar o total da venda, escolha, em bands, a opção *footer*. Adicione-a abaixo do detail, insira um *Object Text* e abra o editor. Observe que ao lado do ícone *Insert Expression* existe um com o símbolo de soma (∑), chamado *Insert Aggregate*. Ao clicar nele, será exibida a janela *Insert Aggregate*, mostrada na **Figura 20**.

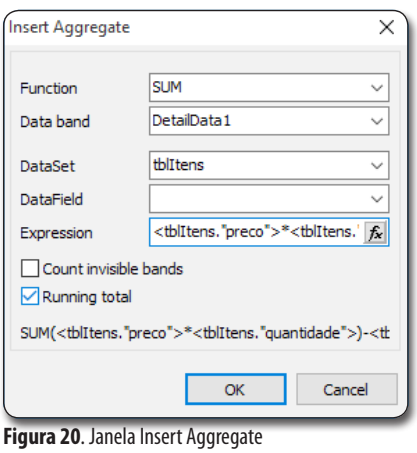

Figura 20. Janela Insert Aggregate

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

A *Insert Aggregate* serve para inserir funções de agregação, como SUM, MAX e MIN. Na opção *Function* são apresentadas as funções disponíveis, em *Data band* se pode escolher qual banda será utilizada, em *DataSet* é definida a query ou o dataSet utilizado, e, em *DataField*, o campo. O *DataField* poderia ser utilizado caso o valor viesse diretamente de um campo da tabela, mas nesse caso, como a soma é resultante de uma expressão, devemos então utilizar o campo *Expression*, cujo valor deve ser o mesmo inserido no campo de total dos itens. Para o exemplo desse artigo preencha os dados como na **Figura 20**, clique em *OK* e retorne ao editor. Ao exibir o preview deve ser apresentado um relatório como o da **Figura 21**.

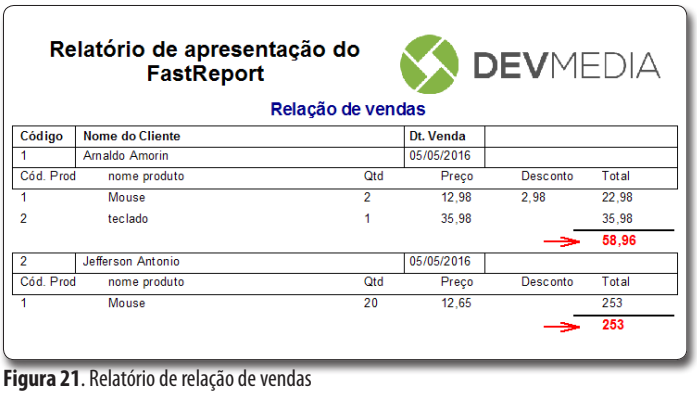

Figura 21. Relatório de relação de vendas

Observe que ainda existe um problema na impressão do relatório: a segunda venda saiu com o total sem as casas decimais, pois o resultado da expressão foi um inteiro. Para ajustar isso, basta abrir o editor desse *text* e, na aba *Format* (veja-a na **Figura 22**), escolher a opção *Number* com o formato desejado.

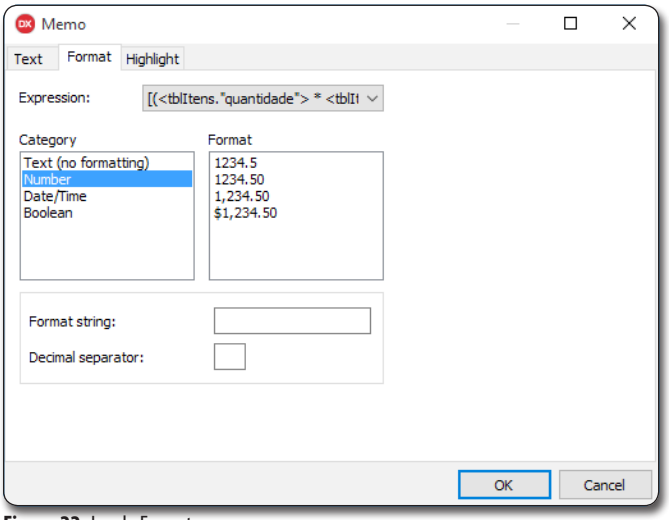

Figura 22. Janela Format

Note que estão disponíveis formatos também para Date/Time e Boolean. No relatório de ficha de clientes essa formatação poderia ser utilizada para alterar a apresentação do valor ao invés do IIF. Note que ele disponibiliza alguns padrões, mas estão em inglês. Para colocar em português, basta que edite o valor para os dados desejados. É permitido que se crie seus próprios formatos, mas infelizmente ao adicionar um formato via design eles não ficam salvos para os próximos usos. Porém, é possível criar e adicionar formatos e categorias através de códigos na chamada do report.

Configurando relatórios

Para configurar o relatório pode ser utilizado o *Object Inspector* para alterar propriedades como *PaperSize* e *Orientation*. Na **Tabela 4** estão descritas as propriedades mais comuns.

Além de estarem presentes no *Object Inspector*, essas configurações também podem ser acessadas pressionando o botão direito sobre a página, e clicando na opção *Edit*. Na janela *Report Settings*, exibida a partir do menu *Report* > *Options...*, pode-se alterar a impressora padrão e, na aba *Description*, adicionar informações de criação do relatório, tais como autor, descrição e até mesmo informações de versão.

Levando DataSets do Delphi para o relatório

Em muitos casos os datasets são carregados e manipulados no Delphi, mas eles também podem ser utilizados no FastReport para preencher os relatórios. Para demonstrar essa funcionalidade será criado um formulário para permitir que o usuário informe um período, a partir do qual o sistema fará uma consulta ao banco a disponibilizará para o FastReport.

Crie um novo formulário e nomeie-o como *frmGetPeriodo*. Esse formulário terá dois componentes *TDateTimePicker*, um botão de ok e outro de cancelar, como na **Figura 23**.

Altere a propriedade *ModalResult* do botão *Cancelar* para *mrCancel* e a do botão *Ok* para *mrOk*. Na unit, adicione um record de nome *TPeriodo* como na **Listagem 4**.

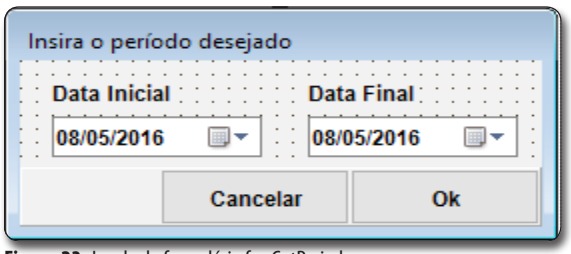

Figura 23. Janela do formulário frmGetPeriodo

Listagem 4. Record TPeriodo

```
01 TPeriodo = record
02   DataIni : TDate;
03   DataFin : TDate;
04 end;
```

Esse record será utilizado para facilitar o retorno de dois parâmetros em uma função. Crie agora o método *GetPeriodo* fora da classe e implemente-o como na **Listagem 5**. Esse método será responsável por chamar a tela de período e validar se os dados foram devidamente preenchidos

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Nome	Descrição
BackPicture	Permite inserir uma imagem de fundo no relatório.
BackPicturePrintable	Determina se a imagem de fundo será também impressa.
BackPictureVisible	Define a visibilidade da imagem de fundo.
Margin	Permite alterar cada uma das margens do relatório (BottonMargin, LeftMargin, RightMargin, TopMargin).
Columns	Determina a quantidade de colunas do relatório.
Duplex	Permite configurar as opções de Duplex.
Orientation	Define a orientação da página (Paisagem ou Retrato).
PaperHeight	Define a altura da página.
PaperWidth	Define a largura da página.
PaperSize	Define o tamanho do papel, que por padrão é A4.
PrintIfEmpty	Define se o relatório será impresso mesmo sem informações.
TitleBeforeHeader	Define se o title será impresso antes do header.

Tabela 4. Propriedades da página

Listagem 5. Método GetPeriodo

```
01 function GetPeriodo : TPeriodo;
02 Var FrmGetPeriodo : TFrmGetPeriodo;
03 begin
04  FrmGetPeriodo := TFrmGetPeriodo.Create(Application);
05  try
06   Result.DataIni := -1;
07   Result.DataFin := -1;
08   If (FrmGetPeriodo.ShowModal = mrOk) Then
09   Begin
10    if (FrmGetPeriodo.DataFin.Date < FrmGetPeriodo.DataIni.Date) then
11     Begin
12      ShowMessage('A data final deve ser maior que a data inicial);
13      Result := GetPeriodo();
14     End
15    else
16     begin
17      Result.DataIni := FrmGetPeriodo.DataIni.Date;
18      Result.DataFin := FrmGetPeriodo.DataFin.Date;
19     end;
20   End
21  finally
22   FreeAndNil(FrmGetPeriodo);
23  end;
24 end;
```

Nas linhas 6 e 7, os valores para o *DataIni* e *DataFin* são setados como -1, facilitando identificar se esses dados foram preenchidos ou se foi pressionado o botão cancelar. Na linha 8, é verificado se a chamada ao *ShowModal* resultou em um *mrOk*. Caso o valor resultante desse teste seja verdadeiro, será efetuado um outro teste, na linha 10, para verificar se a data final é menor que a data inicial. Se a data inicial for maior que a data final, uma mensagem é exibida e o método *GetPeriodo* é chamado novamente, assim obrigando o correto preenchimento dos campos ou o cancelamento. Ao passar por todos os testes, os valores serão repassados para o *result*.

Em seguida, retorne ao formulário principal, adicione um novo menu chamado "Estoque" e um submenu de nome "Relação de movimento por período". Adicione ao formulário um *TFDConnection* e configure a conexão com o banco de dados. Em seguida, adicione um *TFDQuery*, nomeie-o como *QryMovimentacaoEstoque* e preencha seu SQL como na **Listagem 6**. Em *params* altere o *DataType* dos dois parâmetros para *ftDate*.

Para que essa query possa ser lida pelo *TfrxReport*, adicione um *TfrxDBDataset* e renomeie-o para *LinkQryMovimentacaoEstoque*. Ele será responsável por fazer o link do *DataSet* com *TfrxReport*.

No editor de relatório, acesse o menu *Report > Data...* para que seja aberta a janela *Select Report Datasets*. Observe que o *TfrxDBDataset* criado no Delphi está presente nessa lista. Para adicionar o DataSet ao projeto basta marcar o checkbox, e ele passará a ficar disponível no *Data Tree*. Para criar o relatório de movimentação basta arrastar as colunas desejadas e seguir os passos dos relatórios anteriores.

Ao concluir o desenvolvimento do relatório, retorne ao formulário principal e, na chamada do relatório, adicione o código da **Listagem 7**.

Listagem 6. SQL de histórico do estoque

```
01 SELECT *
02 FROM PRODUTOHISTORICOESTOQUE MOV
03 LEFT JOIN PRODUTO PROD ON
04              (PROD.PRODUTOID = MOV.PRODUTOID)
05 WHERE DATA BETWEEN :DATAINI AND :DATAFIN
```

Listagem 7. Método de relação de movimento por período

```
01 procedure TForm1.Relaodemovimentoporperodo1Click(Sender: TObject);
02 Var Perido : TPeriodo;
03 begin
04  Perido := GetPeriodo();
05
06  if (Perido.DataIni = -1 ) then
07   Exit;
08
09  with QryMovimentacaoEstoque do
10  Begin
11   Close();
12   ParamByName('DataIni').AsDate := Perido.DataIni;
13   ParamByName('DataFin').AsDate := Perido.DataFin;
14   Open();
15  End;
16
17  frxReportMovimentacaoEstoque.ShowReport();
18 end;
```

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Na linha 4 é feita a chamada ao método *GetPeriodo*, e seu resultado é armazenado na variável *Periodo*. Na linha 6 é feito um teste para verificar se a data inicial está preenchida. Em seguida, a query é fechada, os parâmetros de *DataIni* e *DataFin* recebem os valores das variáveis de mesmo nome, a query é aberta, e, por fim, na linha 17, é feita a chamada ao método *ShowReport*.

Highlight

O Highlight é um recurso que permite que se crie estilos baseados em condições, dispensando, por exemplo, o uso de códigos. Um exemplo de sua funcionalidade é para locais onde valores maiores que zero devem ser apresentados em azul e os menores que zero apresentados em vermelho.

Para testarmos esse recurso, retorne à edição do relatório de relação de movimento por período. No campo quantidade, abra o editor, vá na aba *Highlight*, e clique em *Add*. Em expression, altere o valor da expressão para *Value > 0*, e, após determinar a expressão, clique na opção *fonte* e altere a cor da fonte para azul.

Adicione uma nova condição onde o *Value* é menor que 0 (*Value < 0*) e altere a cor da fonte para vermelho. Depois, clique em *OK*, execute a aplicação e faça uma consulta de preferência em um período em que exista uma movimentação com valor negativo. Esse recurso também é muito útil quando se tem condições para apresentação de um determinado campo, permitindo apresentar ou não um campo, mudar sua cor de fundo, etc.

Wizard

O FastReport disponibiliza um wizard para facilitar a criação de relatórios simples e padrões, que pode ser acessado pelo menu *File > New*. A tela *New Item* será apresentada, onde escolheremos a opção *Standard Report Wizard*. A primeira tela do wizard permite a seleção do dataset, com duas possibilidades: se já existir um dataset vindo do Delphi, ele pode ser selecionado no combobox, porém, caso siga a estrutura dos primeiros relatórios, em que a conexão é feita através do próprio relatório, o wizard auxilia na criação da conexão e das queries.

Para criar uma nova conexão via wizard, clique no botão "New Query", onde será aberto o wizard para conexão. Com a conexão definida, será aberta a tela para inserir o SQL desejado. Os próximos passos são configurar os campos desejados, os grupos, definir o layout da página e o style.

No layout, pode-se definir a orientação ou layout como tabular ou lista, e no style existem alguns estilos pré-definidos. A **Figura 24** ilustra um exemplo de relatório desenvolvido dessa forma.

O wizard é muito útil, porém o design final não é tão elegante. O ideal é que se utilize o wizard inicialmente para facilitar a conexão com o banco de dados e as tabelas, e no final se customize o relatório como desejado.

Disponibilizando exportações

Caso deseje que os relatórios possam ser exportados para um determinado tipo, basta adicionar ao formulário o componente da paleta *Export* correspondente ao tipo desejado.

Ao executar a aplicação e abrir o relatório, a exportação estará disponível no menu *Salvar*, como pode ser visto na **Figura 25**.

No caso da exportação para PDF e envio de e-mail, são criados menus no topo exclusivamente para esses itens. Ao utilizar a opção de envio de e-mail, a janela *Exportar para E-mail* (presente na **Figura 26**) será apresentada. Nessa janela, informações como a conta de e-mail do remetente e do destinatário devem ser preenchidas.

Existe a possibilidade de os dados já virem preenchidos, ou dessa tela (*Exportar para E-mail*) nem ser apresentada. Nesse caso o e-mail é previamente configurado e enviado sem a necessidade de um preenchimento. Também temos a possibilidade de envio automático do relatório ao ser visualizado ou ao se executar uma determinada ação.

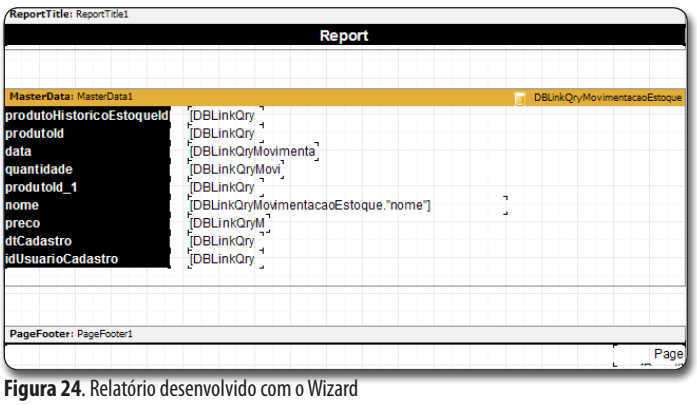

Figura 24. Relatório desenvolvido com o Wizard

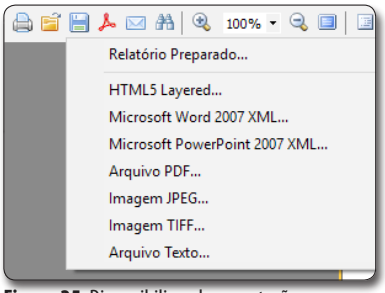

Figura 25. Disponibilizando exportações

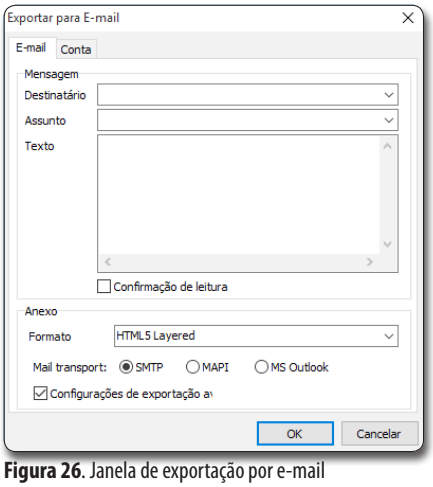

Figura 26. Janela de exportação por e-mail

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Um pouco mais sobre o FastReport

O *TfrxReport* armazena, por padrão, os dados do design do relatório no próprio dfm, fazendo com que esse arquivo fique maior que o normal caso o seu número de relatórios seja grande. Outra desvantagem dessa prática é dificultar a distribuição de relatórios de forma independente. Para que isso não ocorra, basta alterar para *false* a propriedade *StoreInDFM*, fazendo com que os relatórios tenham que ser distribuídos de forma externa ao sistema. Após fazer isso, para carregar o relatório desejado, devemos utilizar o comando *LoadFromFile*. Uma vantagem de se distribuir relatórios de forma independente é que se pode ter apenas um *TfrxReport* para abrir quantos relatórios desejar, basta fazer o *load* do relatório desejado.

Implementações como a criação de um formulário que precisa de informações do usuário, como um período, também podem ser feitas através do FastReport sem a necessidade de alterar o código no Delphi, utilizando, para isso, o *ShowMessage* e o *InputBox* através da área de código do FastReport.

O FastReport conta ainda com outros recursos bem interessantes, como integração de variáveis do *TfrxReport* com o código Delphi, o que permite que variáveis criadas no *TfrxReport* possam ser alteradas no Delphi e que variáveis possam ser criadas no Delphi e enviadas ao *TfrxReport*.

Em situações em que se deseja permitir que o usuário possa alterar os seus relatórios, deve-se adicionar o componente *TfrxDesigner* no form do *TfrxReport* que se deseja permitir edição. Ao fazer isso, o usuário poderá fazer uso do botão "Editar página" para fazer alterações em tempo de execução. Caso o relatório esteja sendo carregado de forma externa, basta que ele seja salvo no mesmo local, sobrescrevendo o anterior, e todas as alterações feitas pelo o usuário estarão disponíveis nas próximas visualizações. Dessa forma, pode-se criar um módulo de relatórios exclusivo do cliente, no qual os relatórios são carregados de uma determinada pasta pré-definida pelo usuário e todos os relatórios salvos nessa pasta serão disponibilizados pelo sistema para o usuário visualizar ou editar, aumentando a customização do software e permitindo que alterações simples não precisem mais ser repassadas ao desenvolvimento, podendo ser feitas pelo próprio cliente.

A impressão de código de barras também é facilitada com o uso do FastReport, que contém um grande número de tipos de códigos de barras disponíveis para uso. Assim, o desenvolvedor não tem que se preocupar com montagem de códigos de barras, basta passar os dados para o report e ele se encarregará de fazer a montagem.

Nesse artigo vimos que poucas vezes a codificação foi necessária, mostrando que o FastReport é muito simples de usar. Além disso, muitas coisas já estão prontas e não precisam ser recriadas, como o caso das variáveis de sistemas (data, contador de página etc.).

Muitas das ferramentas disponíveis no mercado para desenvolvimento de relatórios ou não disponibilizam recursos suficientes para desenvolvimentos mais elaborados, ou são necessárias muitas alterações para a implementação de algo que com o FastReport se torna bem mais simples. Portanto, conhecer bem essa ferramenta é fundamental para passar a utilizá-la e, com isso, obter maior produtividade e oferecer aos clientes maiores opções de relatórios, que normalmente representam elementos importantes dento do sistema.

Autor

Gutierry Antonio
gutierrydsn@hotmail.com

Entusiasta de Big Data e Data Warehouses, atua como Engenheiro de Software e DBA MySQL. É graduado em Sistemas de Informação e Análise de Sistemas, e atualmente cursa especialização em Engenharia de Sistemas. Possui experiência em desenvolvimento com Delphi, Ruby on Rails e com bancos de dados Firebird e MySQL, além de possuir conhecimento em Java, C#, C++ e nos bancos de dados SQL Server e Oracle, bem como em AWS.

Links:

FastReport: Site do produto
https://www.fast-report.com

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Multicamadas e REST com DataSnap

Como construir uma aplicação REST em três camadas com DataSnap

Atualmente, quando se pensa no desenvolvimento de software, dois modelos principais podem ser utilizados: o tradicional modelo cliente/servidor ou o modelo multicamadas. A primeira opção consiste somente de duas camadas: o servidor de banco de dados e a aplicação cliente. Nesse contexto, todas as regras de negócio podem ficar centralizadas tanto na primeira quanto na segunda camada (ou em ambas), e a aplicação cliente acessa diretamente a base de dados. Já no modelo multicamadas existe uma porção de software adicional, que é chamada de servidor de aplicação. Essa camada é responsável por armazenar todas as regras de negócio, e é ela quem interage diretamente com a base de dados, ou seja, nesse modelo a aplicação cliente não tem acesso direto ao servidor de banco de dados. Em suma, a aplicação cliente acessa o servidor de aplicação, enquanto que o servidor de aplicação é quem acessa diretamente a base de dados.

A implementação dessa arquitetura de software possibilita uma série de vantagens, como o balanceamento de carga, que permite definir o número máximo de conexões que um servidor pode suportar, e, quando atingido o limite todo o novo tráfego é automaticamente direcionado para servidores menos ocupados. Algumas outras vantagens desse modelo são: centralização das regras de negócio no servidor (não existem comandos SQL na aplicação cliente); facilidade de redistribuição da aplicação cliente; economia de licenças de acesso a bases de dados; economia de conexões no servidor e escalabilidade.

Uma dúvida muito comum dos desenvolvedores, principalmente na construção de aplicativos móveis, é como definir a arquitetura correta de um app que acessa uma base de dados remota. Nesse contexto, não é uma prática muito comum utilizar o modelo cliente/servidor assim como feito em aplicações desktop, ou seja, fazer uso de componentes de acesso direto a dados conectando em bases de dados remotas. Em aplicativos móveis, deve-se usar componentes de acesso direto aos dados (TFDConnection ou TFDQuery, por exemplo) quando a base de dados é local e está gravada no próprio aparelho, o que normalmente ocorre utilizando o SQLite. Porém, quando há a necessidade de acesso externo, o recomendável é a utilização de um servidor de aplicação.

Fique por Dentro

A utilização do modelo cliente/servidor para o desenvolvimento de software talvez seja o mais largamente utilizado por desenvolvedores. Porém, quando desenvolvemos aplicações mobile, muitas vezes esse modelo não é o mais adequado, pois muitos recursos específicos do servidor precisariam ser desenvolvidos manualmente; fato que pode deixar as aplicações pesadas e difíceis de evoluir. Por isso, é importante o conhecimento sobre como construir arquiteturas multicamadas no Delphi que utilizam os conceitos de REST e tráfego de arquivos JSON pela rede, garantindo maior flexibilidade e escalabilidade às aplicações.

Baseado nisso, o objetivo do presente artigo é mostrar como construir uma arquitetura multicamadas utilizando o Delphi 10.1 Berlin com o SQL Server como o sistema gerenciador de banco de dados. Será mostrado como construir as três camadas desse modelo, partindo da criação da base de dados e implementando também o servidor e a aplicação cliente. O intuito principal é apresentar como a aplicação cliente acessa dados remotos de datasets e incorpora os registros em um ListView, utilizando para isso componentes TFDMemTable. Serão construídos três exemplos de métodos remotos: um para buscar na base de dados a pessoa com o maior salário e retornar o valor; uma pesquisa de pessoas por nome; e, por fim, como trafegar datasets mestre/detalhe pela rede e visualizar seus dados no ListView. Para isso, será utilizado o conceito REST bem como o tráfego de arquivos JSON.

Criando a base de dados

O primeiro passo é a criação da camada de banco de dados da nossa aplicação e, para isso, a **Listagem 1** apresenta os comandos SQL para a criação da base de dados e a definição das tabelas no SQL Server. Na linha 1 encontra-se o comando para a criação

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

do banco, enquanto que entre as linhas 2 e 6 está localizado o comando que define a tabela de funcionários, a qual possui um campo chave primária autoincremento (*idfuncionario*), um nome e o valor do salário. Em seguida, entre as linhas 7 e 12 está o script para gerar a tabela de endereços dos funcionários, que diz respeito ao relacionamento mestre-detalhe (*master-detail*) entre ambas. Essa tabela também contém uma chave primária autoincremento (*idfuncionario_endereco*), a chave estrangeira (*idfuncionario*), o nome do logradouro e o número.

Listagem 1. Scripts de criação da base de dados.

```
01 create database empresa;
02 create table funcionarios (
03   idfuncionario int primary key not null identity,
04   nome varchar(50) not null,
05   salario float not null
06 );
07 create table funcionario_enderecos (
08   idfuncionario_endereco int primary key not null identity,
09   idfuncionario int not null foreign key references funcionarios (idfuncionario),
10   logradouro varchar(50),
11  numero char(10)
12 );
```

A seguir, a **Listagem 2** apresenta os scripts para inserção de dados nas tabelas: entre as linhas 1 e 4 são inseridos alguns funcionários e entre as linhas 5 e 11 seus respectivos endereços são incluídos. Por exemplo, o funcionário de nome "Ana" (*idfuncionario = 3*) possui três endereços, enquanto que a "Jaqueline" (*idfuncionario = 4*) apresenta somente um endereço.

Listagem 2. Scripts para inserção de dados.

```
01 insert into funcionarios (nome, salario) values ('Pedro', 5000);
02 insert into funcionarios (nome, salario) values ('Marcos', 2000);
03 insert into funcionarios (nome, salario) values ('Ana', 8000);
04 insert into funcionarios (nome, salario) values ('Jaqueline', 3000);
05 insert into funcionario_enderecos (idfuncionario, logradouro, numero)
     values (1, 'Rua Joaquim Nabuco', 76);
06 insert into funcionario_enderecos (idfuncionario, logradouro, numero)
     values (1, 'Rua Francisco Pereira', 101);
07 insert into funcionario_enderecos (idfuncionario, logradouro, numero)
     values (2, 'Avenida Central', 205);
08 insert into funcionario_enderecos (idfuncionario, logradouro, numero)
     values (3, 'Rua Pedro Vargas', 540);
09 insert into funcionario_enderecos (idfuncionario, logradouro, numero)
     values (3, 'Avenida Central', 48);
10 insert into funcionario_enderecos (idfuncionario, logradouro, numero)
     values (3, 'Rua Pereira', 2);
11 insert into funcionario_enderecos (idfuncionario, logradouro, numero)
     values (4, 'Rua Carlos Chagas', 1);
```

Criando o servidor DataSnap

O primeiro passo para implementar um novo servidor DataSnap é criar uma nova pasta chamada *Servidor*, na qual todos os arquivos relacionados a ele serão gravados. É uma prática comum dividir os projetos cliente e servidor em diretórios diferentes para manter uma melhor organização do projeto.

Acessando o menu *File>New>Other*, na janela *New Items* deve-se selecionar a opção *DataSnap Server* para que as opções de criação dos servidores sejam exibidas do lado direito. Nessa janela existem três opções: *DataSnap REST Application*, *DataSnap Server* e *DataSnap WebBroker Application*. Para esse exemplo utilizaremos a última opção (*DataSnap WebBroker Application*), a qual possibilita uma fácil migração posterior do servidor para rodar em servidores Apache ou IIS (Internet Information Services). Em seguida, é aberto um assistente composto por cinco etapas para a criação do servidor, conforme é mostrado na **Figura 1**.

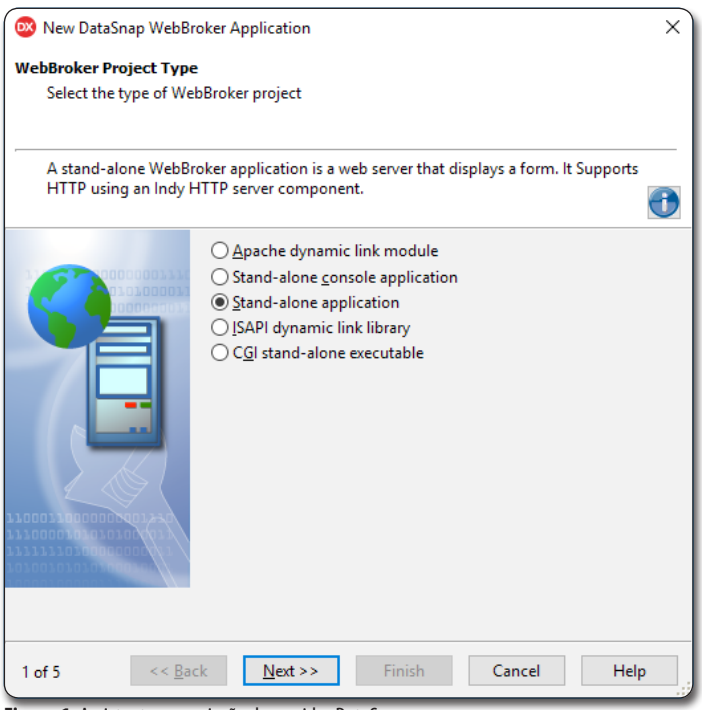

Figura 1. Assistente para criação de servidor DataSnap

Como pode ser observado nessa figura, existem cinco tipos de projetos: a opção *Apache dynamic link module* gera um arquivo para rodar diretamente no Apache no Windows; *Stand-alone console application* não é muito utilizado e o servidor rodará em modo console semelhante ao prompt de comando do Windows; *Stand-alone application* gera um arquivo executável com a extensão ".exe"; *ISAPI dynamic link library* gera uma DLL (Dynamic Link Library) para rodar no IIS; *CGI stand-alone executable* diz respeito à geração de um *cgi* executável para também rodar no IIS. Normalmente, no início do projeto a opção *Stand-alone application* é escolhida pelas facilidades de execução e depuração, porém, esse tipo de servidor não é indicado para a versão final a ser disponibilizada on-line. Isso se deve ao fato de que existe um controle mais eficaz com o IIS ou o Apache, e o uso diretamente do executável no servidor pode atrapalhar o desempenho.

Na segunda etapa do assistente é preciso informar se a aplicação a ser criada será baseada na VCL (Visual Component Library) ou no FireMonkey. Nesse contexto, não faz muita diferença qual das opções escolher, pois para a criação de uma versão Apache

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

ou IIS o formulário não seria utilizado, ou seja, o arquivo principal para criar o servidor seria somente o ServerMethod que será abordado na sequência (neste exemplo utilizaremos a VCL). Na terceira etapa do assistente deve ser informada a porta na qual as comunicações vão ocorrer, sendo geralmente utilizada a porta número 8080 (é importante realizar um teste clicando no botão correspondente para se certificar de que a porta escolhida não está em uso no computador). A escolha da porta nesse momento não é tão importante, pois o número pode ser alterado posteriormente quando o deploy do servidor for realizado. Por fim, ainda na terceira etapa é possível mudar o protocolo para HTTPS, fato que vai adicionar uma nova etapa no assistente para serem informados dados adicionais, principalmente questões relacionadas a certificados. Novamente, não é necessário realizar essas configurações nesse momento, pois é possível fazer manualmente depois.

Na quarta etapa do assistente (*Server Features*) são mostradas algumas configurações de segurança. O checkbox *Authorization* é relativo à utilização de tokens para evitar que alguém invada o sistema e descubra os métodos remotos que estão no servidor. A opção *Sample Methods* gerará dois métodos de teste que são comumente utilizados para verificar se a conexão com o servidor está funcionando. Por fim, a opção *Encryption* refere-se à criptografia dos dados enviados, enquanto que *Compression* é um recurso muito importante para ser utilizado na versão final do servidor, pois ele permite um aumento da velocidade do download dos dados trafegados. Nesse exemplo, nenhuma dessas opções será selecionada.

Por fim, na quinta e última etapa é necessário escolher de qual classe a unit ServerMethods (gerada posteriormente) será descendente, podendo escolher entre *TComponent*, *TDataModule* ou *TDSServerModule*. A primeira opção é uma classe não visual na qual todos os componentes de acesso a dados deverão ser instanciados. A segunda é um DataModule padrão, porém ele é mais para uso interno e não provê métodos do servidor. A última opção (*TDSServerModule*) é a mais utilizada e os métodos do servidor serão salvos nela, podendo inclusive existir vários DSServerModule para um mesmo servidor.

Concluído o assistente, deve-se agora salvar todos os arquivos na pasta *Servidor*. São quatro arquivos a serem salvos: o formulário com os botões *start* e *stop* do servidor (que deve ser nomeado como *frmPrincipalServidor*), um módulo web (*webModule*), o server methods (*metodosServidor*) e o projeto (*Servidor*). O arquivo *webModule* é reponsável por receber as conexões da rede e existe somente um dentro do projeto. Por outro lado, o arquivo *metodosServidor* é semelhante a um DataModule e é nele que programaremos os métodos remotos, bem como adicionaremos todos os componentes de acesso a dados. Dependendo do tamanho da aplicação, é recomendável criar uma unit dessas para cada setor do software, ou seja, um para contas a pagar, outro para contas a receber, outro para estoque e assim por diante. Apesar de ser possível adicionar todas as funções em um único arquivo, o ideal é dividir de acordo com o escopo da aplicação.

Adicionando os componentes de acesso a dados no servidor

O próximo passo é adicionar os componentes de acesso aos dados na unit *metodosServidor*, como mostra a **Figura 2**. É necessário adicionar um TFDConnection (*cnnConexao*), três TFDQuery (*qryMaiorSalario*, *qryFuncionarios* e *qryEnderecos*), um TFDPhysMSSQLDriverLink (*driver*) e um TFDStanStorageJSONLink (*link*). O primeiro componente realizará a conexão do servidor com a base de dados; as queries terão a função de executar as consultas para buscar os registros; o *driver*, como o próprio nome indica, representa o driver de acesso ao SQL Server; enquanto que o componente *link* informa o tipo de dados que será trafegado entre a aplicação servidora e a aplicação cliente, ou seja, arquivos no formato JSON. Dessa forma, quando uma consulta for realizada, os dados provenientes do componente TFDQuery serão empacotados nesse formato para trafegarem na rede.

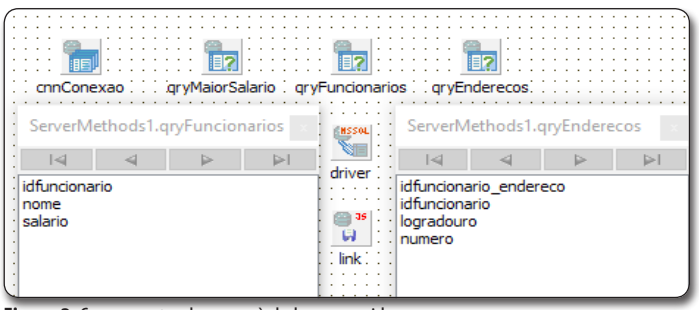

Figura 2. Componentes de acesso à dados no servidor

Clicando duas vezes no componente de conexão (*cnnConexao*) será aberta a janela *FireDAC Connection Editor* (**Figura 3**), na qual deve ser selecionado o *Driver ID* para *MSSQL*, além de preencher o nome *Empresa* na propriedade *Database* e o valor *localhost* na propriedade *Server*. O SQL Server disponibiliza duas opções para seu acesso: a definição de usuário e senha específicos ou então a utilização da segurança integrada com o Windows. Caso se esteja utilizando a primeira opção, os campos *User_Name* e *Password* devem ser preenchidos, e, caso se utilize a segurança integrada, deve-se clicar no botão *Wizard* da **Figura 3** e marcar a opção *Use Trusted Connection*. Além disso, a propriedade *Login Prompt* do FDConnection deve ser marcada como *False* para que a janela de login não seja aberta a toda tentativa de conexão.

O componente *qryMaiorSalario* deve ter a sua propriedade *SQL* preenchida com o seguinte comando:

```
select max(salario) as maior from funcionarios
```

Essa instrução SQL está buscando dentro da tabela de funcionários o valor do maior salário, e será utilizada na sequência para a construção do método que retorna esse valor para a aplicação cliente. Por outro lado, o componente *qryFuncionarios* deve ter o seguinte comando também na propriedade *SQL*, o qual busca todos os funcionários cadastrados na base:

```
select idfuncionario, nome, salario from funcionarios
```


DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

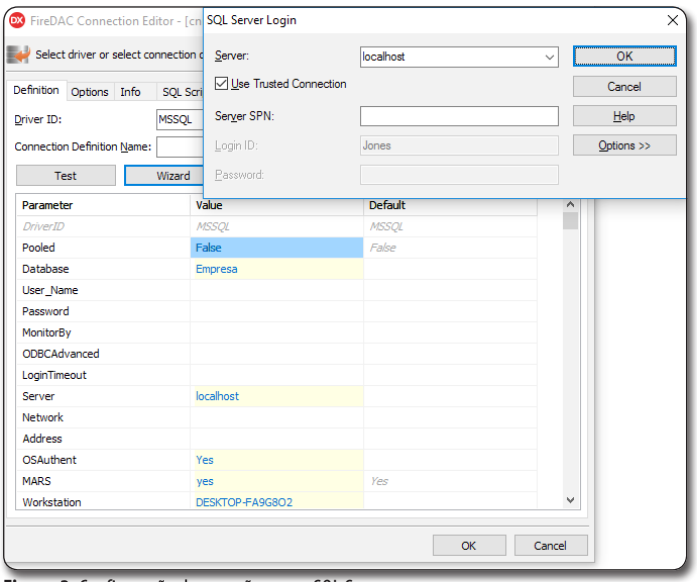

Figura 3. Configuração da conexão com o SQL Server

Por fim, a *qryEnderecos* deve ser preenchida com o comando a seguir, que recupera os endereços dos funcionários cadastrados:

```
select idfuncionario_endereco, idfuncionario, logradouro, numero from funcionario_enderecos
```

Para os dois últimos componentes, deve-se clicar com o botão direito, acessar o *Fields Editor* e logo após escolher a opção *Add All Fields*. Dessa forma, todos os campos estarão nessa lista, conforme mostrado pela **Figura 2**. É também importante salientar que esses três componentes devem estar ligados ao *cnnConexao* por meio da propriedade *Connection*.

Nota

Além do componente TFDStanStorageJSONLink, existem também na paleta FireDAC Links os componentes TFDStanStorageXMLLink e TFDStorageBinLink. Eles são utilizados, respectivamente, para trafegar XML ou arquivos no formato binário entre as aplicações cliente e servidor.

Criando os métodos no servidor

Após todos os componentes de acesso a dados estarem devidamente adicionados e configurados, o próximo passo é a criação dos métodos remotos na unit *metodosServidor*. Para isso, é primeiramente necessário adicionar o namespace *Data.FireDACJSONReflect* na cláusula *Uses* desse arquivo, para que seja possível acessar os métodos para controle dos datasets no formato JSON. A **Listagem 3** apresenta a assinatura dos três métodos a serem codificados nesse exemplo, os quais devem ser digitados na seção *public* do formulário. Declará-los nessa seção quer dizer que eles serão visíveis na aplicação cliente. Caso desejemos que um método não seja acessado no lado cliente, o mesmo deve ser declarado na seção *private* do formulário.

Apertando o conjunto de teclas CTRL+SHIFT+C será gerada a implementação dos três métodos na seção *implementation* do arquivo. A **Listagem 4** apresenta a codificação completa do primeiro deles, na qual pode-se observar que a função não recebe parâmetro e retorna um objeto do tipo TJSONNumber. O objetivo desse método é consultar qual é o maior salário dos funcionários cadastrados e retornar o valor para a aplicação cliente. Para isso, na linha 3 a consulta é fechada para evitar trazer dados do cache do servidor, na linha 4 ela é aberta e consequentemente o seu comando SQL definido anteriormente é executado. Por fim, na linha 5 é criado um novo objeto da classe TJSONNumber que recebe como parâmetro o valor retornado pela consulta no formato *float*.

Listagem 3. Método para retornar o maior salário.

```
01 public
02   function GetMaiorSalario(): TJSONNumber;
03   function GetFuncionarios(Nome: string): TFDJSONDataSets;
04   function GetFuncionariosEnderecos(IdFuncionario: integer): TFDJSONDataSets;
05 end;
```

Listagem 4. Método para retornar o maior salário.

```
01 function TServerMethods1.GetMaiorSalario: TJSONNumber;
02 begin
03   qryMaiorSalario.Active := False;
04   qryMaiorSalario.Open();
05   Result := TJSONNumber.Create(qryMaiorSalario.FieldByName('maior').AsFloat);
06 end;
```

Nota

Tipos de dados como string podem ser utilizados normalmente como retorno em funções DataSnap. Porém, ao criar um método que retorne um tipo de dados real, não acontecerá nenhum erro no servidor ou na compilação, contudo o método simplesmente não será visualizado pela aplicação cliente. Por esse motivo é que na **Listagem 4** foi utilizado um tipo de dado TJSONNumber.

Dando continuidade à implementação do exemplo, a **Listagem 5** apresenta uma função que recebe como parâmetro o nome de um funcionário, realiza uma consulta na base de dados procurando por esse nome e retorna um dataset, o qual é representado pela classe TFDJSONDataSets que pode ser visto na linha 1. Na linha 3 é definida uma constante com o comando SQL, que é o mesmo que foi anteriormente configurado no componente *qryFuncionarios*. É importante enfatizar que a consulta realizada anteriormente foi utilizada somente para termos acesso aos campos da tabela no formato *TField*, os quais utilizaremos posteriormente para a construção da parte cliente da aplicação. Na linha 5 a query é desativada para evitar buscar dados do cache e na linha 6 é criado um objeto do tipo TFDJSONDataSets e atribuído ao retorno da função. Na linha 7 é feita uma estrutura condicional que verifica se o parâmetro *Nome* é ou não vazio, e caso não seja, o comando SQL da linha 3 é concatenado com um *where* que busca os registros de acordo com o nome informado pelo usuário na aplicação cliente. Caso o parâmetro seja vazio, na linha 10 o comando SQL original é carregado na query para buscar todos os registros. Por fim, na linha 11 é chamado o método *ListAdd* da classe *TFDJSON-*

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

DataSetsWriter, que recebe como parâmetro o *Result* da função juntamente com a *qryFuncionarios*. Dessa forma, dentro do *Result* (*TFDJSONDataSets*) serão copiados todos os dados retornados pela *qryFuncionarios*, sem a necessidade de abrir a consulta. Em outras palavras, na linha 11 é que o pacote de dados é enviado para a aplicação cliente.

Listagem 5. Método para retornar os funcionários por nome.

```
01 function TServerMethods1.GetFuncionarios(Nome: string): TFDJSONDataSets;
02 const
03   SQL = 'select idfuncionario, nome, salario from funcionarios';
04 begin
05   qryFuncionarios.Active := False;
06   Result := TFDJSONDataSets.Create;
07   if Nome <> '' then
08     qryFuncionarios.SQL.Text := SQL + ' where nome = ' + QuotedStr(Nome)
09   else
10     qryFuncionarios.SQL.Text := SQL;
11  TFDJSONDataSetsWriter.ListAdd(Result, qryFuncionarios);
12 end;
```

De forma bastante similar, a **Listagem 6** apresenta o código que filtrará um funcionário pela sua chave primária e retornará seus dados juntamente com seus respectivos endereços, caracterizando o relacionamento mestre-detalhe entre as duas tabelas. É possível observar na linha 1 que essa função recebe como parâmetro um inteiro (*IdFuncionario*) e também retorna um objeto do tipo *TFDJSONDataSets*. Nas linhas 3 e 4 estão definidas as constantes com os comandos *select* para realizarem a consulta na base de dados, ou seja, na tabela de funcionários e endereços dos funcionários, respectivamente. Note que no final dos dois comandos há um caractere *%s*, que receberá na sequência o código do funcionário informado pelo usuário na aplicação cliente. Similarmente ao método anterior, nas linhas 6 e 7 as duas queries são desativadas e na linha 8 o *Result* da função recebe uma nova instância da classe *TFDJSONDataSets*. Nas linhas 9 e 10, por meio do comando *Format*, o valor do parâmetro *IdFuncionario* é substituído pelo caractere *%s*, fato que trará somente os registros que correspondam ao código informado na aplicação cliente. O ponto mais importante deste código está nas linhas 11 e 12, nas quais é chamado novamente o método *ListAdd*, uma vez para cada uma das queries. Dessa forma, quando há necessidade de retornar do servidor para a aplicação cliente um ou mais datasets, particularmente comum em aplicações mestre detalhe; basta somente um método. A classe *TFDJSONDataSets* permite que sejam adicionados vários datasets em sua estrutura, sendo que cada um deles utilizará um índice para o diferenciar dos demais. Por exemplo, o dataset com o nome *funcionarios* (linha 11) será o índice 0 (zero), enquanto que o dataset com o nome *enderecos* será o índice 1 (um). Com isso, toda a filtragem dos dados a serem retornados para o cliente é feita no próprio servidor, enviando para a outra camada somente os dados que são efetivamente relevantes.

A última etapa relacionada ao servidor é testar sua execução, o que pode ser observado na **Figura 4**.

Essa janela é criada automaticamente pelo RAD Studio e o botão *Start* deve ser clicado para iniciar o servidor. Adicionalmente, clicando no botão *Open Browser* ou digitando no navegador a URL *http://localhost:8080/* é possível testar se o servidor DataSnap está em execução. Caso esse software esteja rodando em uma outra máquina dentro de uma rede corporativa, ou até mesmo na Internet, bastaria trocar o *localhost* pelo IP da máquina para conseguir acesso.

Listagem 6. Método para retornar funcionários e endereços (mestre detalhe).

```
01 function TServerMethods1.GetFuncionariosEnderecos(IdFuncionario: integer):
     TFDJSONDataSets;
02 const
03   SQLFuncionario = 'select idfuncionario, nome, salario from funcionarios where
       idfuncionario = %s';
04   SQLEndereco = 'select idfuncionario_endereco, idfuncionario, logradouro,
       numero from funcionario_enderecos where idfuncionario = %s';
05 begin
06   qryFuncionarios.Active := False;
07   qryEnderecos.Active := False;
08   Result := TFDJSONDataSets.Create;
09   qryFuncionarios.SQL.Text := Format(SQLFuncionario, [IdFuncionario.ToString]);
10   qryEnderecos.SQL.Text := Format(SQLEndereco, [IdFuncionario.ToString]);
11   TFDJSONDataSetsWriter.ListAdd(Result, 'funcionarios', qryFuncionarios);
12   TFDJSONDataSetsWriter.ListAdd(Result, 'enderecos', qryEnderecos);
13 end;
```

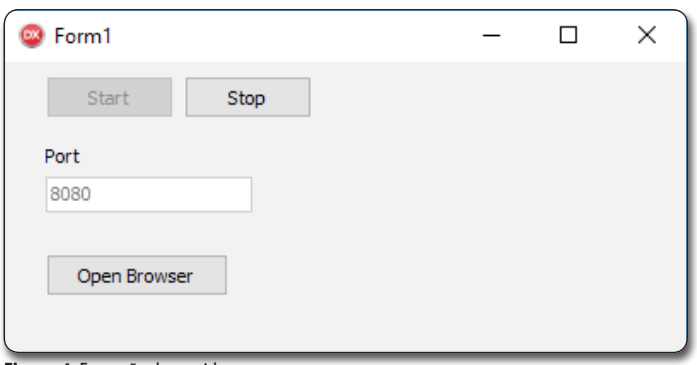

Figura 4. Execução do servidor

Criando o aplicativo cliente

A partir de agora construiremos a parte cliente do projeto, que acessará os métodos definidos no servidor DataSnap. Para isso, um novo projeto multidispositivo deve ser criado no IDE do RAD Studio (*File>New>Multi-Device Application – Delphi*), escolhendo a opção *Blank Application*. Esse novo projeto pode ser definido na mesma instância aberta do Delphi que já contém o projeto servidor, pois trabalharemos com o conceito de *project group* para não ser necessário abrir duas instâncias do Delphi. Para isso, basta clicar com o botão direito na janela e escolher a opção para criação de novo projeto.

Projetos multidispositivos utilizam o FireMonkey, o que permite que a compilação possa ser feita para Windows, Android, iOS e OSX utilizando o mesmo código fonte, fazendo com que o aplicativo funcione tanto em ambientes mobile quanto desktop. A **Figura 5** apresenta a janela da aplicação, que possui três TEdit

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

(*edtFuncionarioNome*, *edtIdFuncionario* e *edtSalario*), três TButton (*btnFuncionariosNome*, *btnFuncionarioEndereco* e *btnSalario*) e três TListView (*lsvFuncionarioNome*, *lsvFuncionario* e *lsvEndereco*). Quando o botão *Funcionário por nome* for clicado, o nome digitado no edit acima dele será passado como parâmetro para o servidor DataSnap, que executará o método da **Listagem 5** e devolverá o dataset, que será atribuído ao primeiro ListView. De forma similar, quando o botão *Funcionário/endereço* for acionado, o valor do *IdFuncionario* será também enviado ao servidor, que executará o código da **Listagem 6** e retornará os dois datasets, que serão atribuídos aos outros dois ListView para caracterizar o relacionamento mestre detalhe entre as tabelas. Por fim, clicando-se no botão *Maior salário* o método da **Listagem 4** será invocado, retornando no edit acima dele o valor do salário.

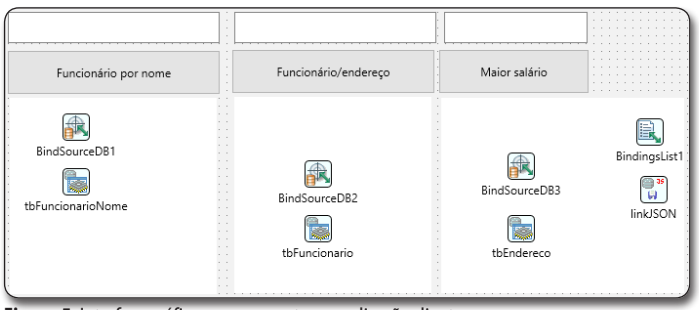

Figura 5. Interface gráfica e componentes na aplicação cliente

Com relação aos componentes não visuais, pode-se observar que existem três componentes *BindSourceDB* e um *BindingList*, os quais serão automaticamente adicionados pelo RAD Studio quando as ligações LiveBindings forem feitas; o que indica que eles não precisam ser adicionados manualmente. Contudo, é necessário adicionar um *TFDStanStorageJSONLink* (*linkJSON*) para permitir que os dados trafeguem no formato JSON, bem como três *TFDMemTable* (*tbFuncionarioNome*, *tbFuncionario* e *tbEndereco*) que estarão associados aos controles ListView.

Nota

O componente TFDMemTable da biblioteca FireDAC é utilizado para gerenciar dados em memória, muito semelhante ao TClientDataSet que é bastante utilizado em conjunto com os componentes DBExpress. Ele possui funcionalidades parecidas com um TClientDataSet, porém é mais rápido.

Todos os arquivos desse novo projeto devem ser salvos em uma pasta chamada *Cliente*, para manter a organização da camada servidora e cliente de nosso exemplo (nesse momento serão salvos somente o formulário - *PrincipalFrm* e o projeto – *Cliente*). Conforme mencionado anteriormente, uma prática bastante comum para trabalhar com dois projetos no RAD Studio é utilizar o conceito de *Project Group* (grupo de projetos), para que assim seja utilizada somente uma instância do Delphi aberta com os projetos referentes ao cliente e ao servidor. A **Figura 6** apresenta essa estrutura na janela *Project Manager*, na qual o nome *DataSnap* refere-se ao nome do grupo e possui a extensão *.groupproj*.

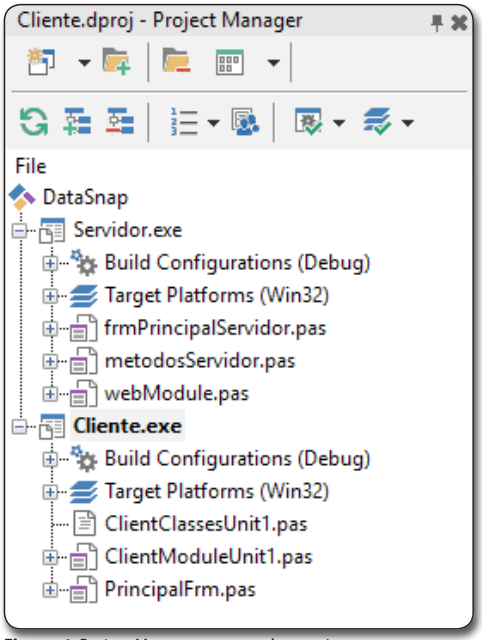

Figura 6. Project Manager mostrando o project group

É necessário agora criar o acesso ao servidor DataSnap, acessando o menu *File>New>Other* e em *DataSnap Server* escolhendo a opção *DataSnap REST Client Module*, conforme é mostrado na **Figura 7**. Confirmando a escolha dessa opção, será exibido um assistente de três passos, que é bastante similar à criação do servidor. Na primeira etapa deve-se escolher *Local server* caso estejamos trabalhando com servidor local, e na segunda etapa é necessário marcar o radiobutton *WebBroker stand alone server*, já que esse foi o tipo de servidor criado anteriormente. Por fim, na última etapa nenhuma informação precisa ser modificada, basta apenas clicar no botão *Test Connection* para verificar se o servidor está em execução. É importante salientar que não é possível criar o módulo de conexão cliente caso o servidor não esteja rodando.

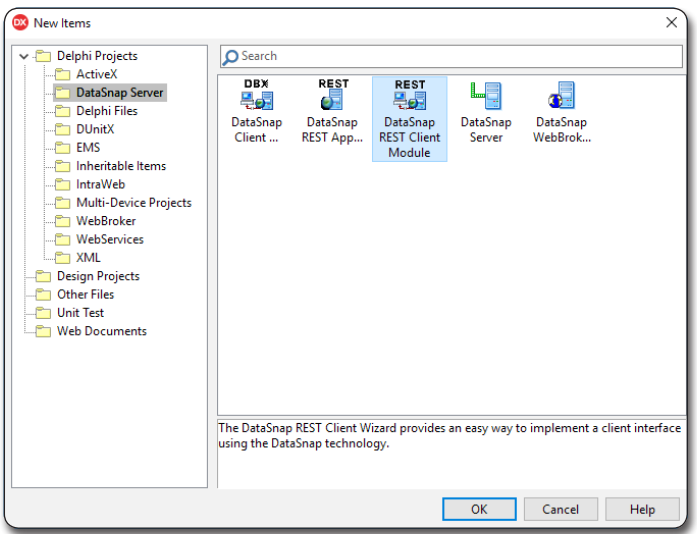

Figura 7. Criação do cliente DataSnap

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Após o término do assistente, serão gravados dois novos arquivos: *ClientModuleUnit1* e *ClientClasses1*, os quais devem ser salvos também na pasta *Cliente*. O arquivo *ClientClasses* também pode ser chamado de proxy, pois é uma classe que abstrai os métodos do servidor e fará a interface entre esse e o cliente. A **Listagem 7** mostra a seção *public* desse arquivo, na qual é possível observar entre as linhas 5 e 7 que os métodos definidos no servidor foram copiados. É importante enfatizar que esses métodos são somente uma interface para acesso ao servidor, e a real implementação deles ainda está mantida no próprio servidor de aplicação.

Listagem 7. Interface dos métodos do servidor no cliente.

```
01 public
02   constructor Create(ARestConnection: TDSRestConnection); overload;
03   constructor Create(ARestConnection: TDSRestConnection;
       AInstanceOwner: Boolean); overload;
04   destructor Destroy; override;
05   function GetMaiorSalario(const ARequestFilter: string = ''): TJSONNumber;
06   function GetFuncionarios(Nome: string; const ARequestFilter: string = '':
       TFDJSONDataSets;
07   function GetFuncionariosEnderecos(IdFuncionario: Integer;
       const ARequestFilter: string = ''): TFDJSONDataSets;
08 end;
```

O arquivo *ClientModuleUnit1*, por sua vez, é a unit que efetivamente se conecta ao servidor e invoca os métodos para execução. A **Figura 8** mostra o componente *DSRestConnection*, que é o responsável pela conexão. Toda vez que um novo método é adicionado no servidor, o mesmo deve ser reiniciado e a opção *Generate DataSnap client classes* deve ser executada conforme mostrado na figura, para que todos os novos métodos sejam gerados na unit *ClientClassesUnit1*.

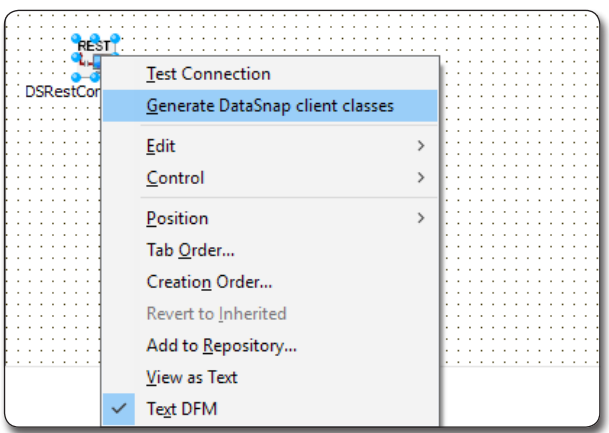

Figura 8. DSRestConnection

Chamada dos métodos do servidor no cliente

O primeiro passo para invocar os métodos do servidor em nossa aplicação cliente é fazer adicionar na seção *Uses* dois namespaces e uma unit no formulário principal (*PrincipalFrm*), conforme mostrado na **Listagem 8**. O *Data.FireDACJSONReflect* terá a função de ler os datasets no formato JSON, a *ClientModuleUnit1* contém a propriedade para que possamos acessar os métodos remotos, e o *System.JSON* permitirá acesso aos objetos do tipo *TJSONNumber* que será utilizado para o retorno do maior salário.

O código da **Listagem 9** deve ser implementado no evento *OnClick* do botão *btnSalario*. Na linha 3 é criada uma variável do tipo *TJSONNumber*, que é o mesmo tipo do retorno do método criado no servidor na **Listagem 4**. Na linha 5 é chamada a unit *ClientModule1*, a qual possui a propriedade *ServerMethods1Client* que permite acesso à interface dos métodos remotos. Ainda na linha 5, a variável *Salario* receberá o valor de retorno do método remoto, que é copiado para o *edtSalario* na linha 6. Dessa forma, a chamada ao primeiro método utilizando o servidor DataSnap está concluída.

Listagem 8. Uses da aplicação cliente.

```
1 implementation
2 uses Data.FireDACJSONReflect, ClientModuleUnit1, System.JSON;
```

Listagem 9. Chamada ao método do maior salário.

```
01 procedure TForm2.btnSalarioClick(Sender: TObject);
02 var
03   Salario: TJSONNumber;
04 begin
05   Salario := ClientModule1.ServerMethods1Client.GetMaiorSalario();
06   edtSalario.Text := Salario.ToString;
07 end;
```

Em seguida, a **Listagem 10** apresenta a chamada ao método para pesquisar o funcionário pelo nome (**Listagem 5**), que deve ser programado no evento *OnClick* do botão *btnFuncionarioNome*. Na linha 3 é definida a variável *dsFuncionarios* do tipo *TFDJSONDataSets*, o mesmo do retorno do método no servidor. Na linha 5 o método no servidor é invocado por meio da interface *ClientModule1* e é passado por parâmetro o texto que o usuário informar no *edtFuncionarioNome*. Dessa forma, o *dsFuncionarios* conterá todos os registros retornados pelo método remoto. A linha 6 verifica se existe algum dataset retornado por meio do método *GetListCount*, bastante semelhante ao *RecordCount* comumente utilizado em outros tipos de datasets. A diferença é que o RecordCount informa a quantidade de registros que a consulta retornou, enquanto que o *GetListCount* faz a contagem de quantos datasets estão presentes na variável *dsFuncionarios*. Nesse caso, se existir um dataset, quer dizer que pelo menos um registro foi carregado. Como esses dados serão populados em um ListView por meio de uma tabela em memória, na linha 8 o *tbFuncionarioNome* é primeiramente desativado para que então na linha 9 seja utilizado o método *AppendData* para populá-lo. Ao contrário dos métodos do servidor, a classe *TFDJSONDataSetsReader* é usada para ler os dados do objeto *dsFuncionarios*, e o parâmetro 0 (zero) diz respeito ao índice do primeiro dataset retornado. Nesse método, esse é o único índice existente já que se trata apenas de um dataset. Por fim, na linha 10 o memory table é ativado para que os registros sejam exibidos.

É necessário ainda fazer a vinculação via Live Bindings do memory table (*tbFuncionarioNome*) com o ListView (*lsvFuncionarioNome*). Para que isso seja possível, a tabela em memória necessita dos campos para vinculação, os quais estão localizados no *Fields*

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Editor do componente *qryFuncionarios*, conforme pode ser observado na **Figura 2**. Em conformidade com o que foi mencionado anteriormente, esses campos foram adicionados no servidor para que possamos copiá-los e colá-los no *Fields Editor* do memory table. Dessa forma, deve-se voltar na aplicação servidora e abrir o *Fields Editor* da query (botão direito>*Fields Editor*), copiar os campos (CTRL+C), abrir o *Fields Editor* no memory table (duplo clique no componente) e colá-los (CTRL+V). Após isso, pode-se vinculá-lo ao ListView de acordo com a **Figura 9**, ou seja, somente o campo *nome* ligado ao *Item.Text*.

Listagem 10. Chamada ao método de pesquisa por nome.

```
01 procedure TForm2.btnFuncionariosNomeClick(Sender: TObject);
02 var
03   dsFuncionarios: TFDJsonDataSets;
04 begin
05   dsFuncionarios := ClientModule1.ServerMethods1Client.GetFuncionarios
       (edtFuncionarioNome.Text);
06   if TFDJSONDataSetsReader.GetListCount(dsFuncionarios) = 1 then
07   begin
08     tbFuncionarioNome.Active := false;
09     tbFuncionarioNome.appendData(TFDJSONDataSetsReader.GetListValue
       (dsFuncionarios, 0));
10     tbFuncionarioNome.active := true;
11   end;
12 end;
```

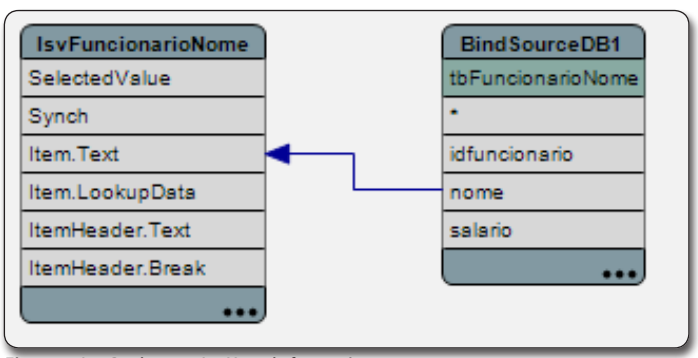

Figura 9. LiveBindings no ListView de funcionários

Nota

Após a vinculação via Live Bindings é que os componentes BindingSourceDB e BindingsList mostrados na **Figura 5** serão adicionados ao formulário.

De forma bastante similar, a **Listagem 11** apresenta a codificação necessária para carregar os dois datasets (de funcionários e de endereços) nos outros dois ListViews, a qual deve ser implementada no evento *OnClick* do botão *btnFuncionarioEndereco*. Na linha 3 é declarada a variável *dsFuncionariosEnderecos*, que armazenará os dois datasets que serão retornados pelo servidor. Na linha 5, por meio da interface *ClientModule1* o método remoto *GetFuncionariosEnderecos* é invocado, passando como parâmetro o código do funcionário que o usuário informará no edit *edtIdFuncionario*. Na linha 6 é feita a verificação para certificar-se de que estão sendo retornados dois datasets, um para o mestre e outro para o detalhe. Nas linhas 8 e 9 os dois memory tables são fechados para que então, na linha 10, o *tbFuncionario* receba o índice 0 (zero) do objeto *dsFuncionariosEnderecos*, enquanto que na linha 11 o componente *tbEndereco* recebe o índice 1 (um). Dessa maneira, tabelas mestre/detalhe são retornadas do servidor para a aplicação cliente, utilizando uma única variável. E o mais importante: somente será feita uma única chamada ao servidor; a qual encapsulará os dados no formato JSON e devolverá para o aplicativo cliente todos os datasets por meio da classe *TFDJSONDataSets*. Por fim, nas linhas 12 e 13 as tabelas em memória são ativadas para que se possa exibir os dados.

Nota

Na **Listagem 11** estamos assumindo que um registro master sempre terá pelo menos um registro detalhe, por isso foi utilizado o valor "= 2" na linha 6. Caso exista um mestre sem detalhe, as linhas abaixo da estrutura condicional if não serão executadas.

Listagem 11. Chamada ao método para trazer as tabelas mestre detalhe.

```
01 procedure TForm2.btnFuncionarioEnderecoClick(Sender: TObject);
02 var
03   dsFuncionariosEnderecos: TFDJsonDataSets;
04 begin
05   dsFuncionariosEnderecos := ClientModule1.ServerMethods1Client
       .GetFuncionariosEnderecos(edtIdFuncionario.Text.ToInteger());
06   if TFDJSONDataSetsReader.GetListCount(dsFuncionariosEnderecos) = 2 then
07   begin
08     tbFuncionario.Active := False;
09     tbEndereco.Active := False;
10     tbFuncionario.AppendData(TFDJSONDataSetsReader.GetListValue
       (dsFuncionariosEnderecos, 0));
11     tbEndereco.AppendData(TFDJSONDataSetsReader.GetListValue
       (dsFuncionariosEnderecos, 1));
12     tbFuncionario.Active := True;
13     tbEndereco.Active := True;
14   end;
15 end;
```

Similarmente ao passo anterior, deve-se copiar os campos do *Fields Editor* dos componentes *qryFuncionarios* e *qryEnderecos* para o *Fields Editor* dos componentes no cliente *tbFuncionario* e *tbEndereco*, respectivamente. Após isso, pode-se efetuar a ligação via Live Bindings entre esses componentes e os respectivos ListViews, conforme mostrado na **Figura 10**. Para que os dois ListViews apresentem os mesmos campos da figura, deve-se mudar sua propriedade *ItemAppearance>ItemAppearance* para *ListItemRightDetail*, para que seja adicionada a opção *Item.Detail* no LiveBindings. Com isso, o resultado final da execução será semelhante ao mostrado na **Figura 11**.

Nota

Caso o projeto do servidor esteja inacessível e não seja possível copiar os campos pelo Fields Editor, uma alternativa é fazer um processamento via código no objeto TFDJSONDataSets e popular o ListView também via código. Consulte a seção **Links** para mais informações.

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

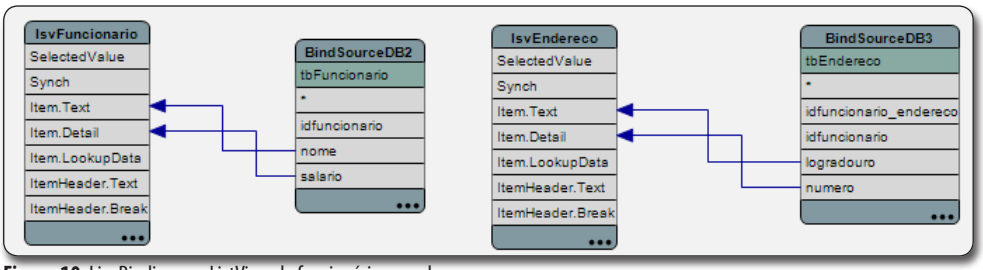

Figura 10. LiveBindings no ListView de funcionários e endereços

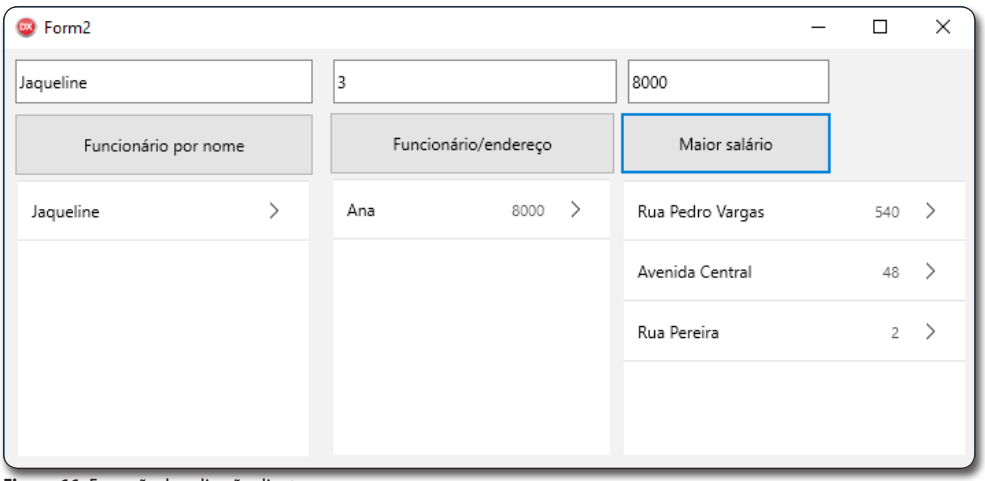

Figura 11. Execução da aplicação cliente

Nota

A chamada dos métodos do servidor nas **Listagens 10** e **11** realizam o download dos dados do servidor, ou seja, em um dispositivo móvel é a execução desses métodos que consumirá os recursos de rede, possivelmente utilizando o plano de dados do aparelho. Por esse motivo, é de extrema importância otimizar a quantidade de registros que serão retornados. Quando o memory table é preenchido, já não existe mais a conexão com o servidor de aplicação.

A **Figura 11** apresenta um exemplo de execução da aplicação no Windows, na qual é possível observar que à esquerda foi pesquisado o nome "Jaqueline" e no ListView abaixo o registro correspondente foi listado. Na parte central da figura, foi digitado o número 3, que corresponde à funcionária "Ana" e que possui três endereços, conforme o terceiro ListView. Por fim, clicando no botão *Maior salário*, é copiado o valor "8000" para o terceiro edit, que corresponde ao maior salário encontrado na base de dados.

Com isso, nosso projeto multicamadas já está funcional e com vários recursos implementados, os quais são comumente utilizados em aplicações tanto mobile quanto desktop ou web. Dessa forma, há uma divisão entre as três principais camadas do software, ou seja, o banco de dados, o servidor de aplicação e o cliente.

Como pôde ser observado, a camada cliente nunca acessa diretamente a base de dados como no clássico modelo cliente/servidor, mas, ao invés disso, ela acessa um servidor de aplicação que disponibiliza as interfaces para acesso aos métodos remotos.

Outra alternativa muito utilizada no desenvolvimento móvel é fazer uso de uma base de dados local (SQLite, por exemplo) instalada no próprio aparelho e outra base de dados remota que ficaria na sede de uma determinada empresa, por exemplo. Suponhamos que um vendedor realizará algumas vendas diretamente pelo aparelho, mas não existe conexão à Internet disponível. Nesse contexto, os dados da venda podem ser armazenados localmente no aparelho, e depois que o celular estiver conectado, pode haver um mecanismo de sincronização entre as duas bases de dados. Para implementar esses recursos, também são utilizadas as técnicas abordadas nesse artigo.

Autor

Jones Granatyr

Doutorando em Informática bolsista CAPES e Mestre em Ciência da Computação bolsista CNPq, ambos na área de Inteligência Artificial. É fundador do portal IA Expert e trabalha em projetos de pesquisa relacionados a área de Inteligência Artificial, tais como Sistemas Especialistas, Mineração de Dados, Mineração de Textos e Sistemas Multiagente.

Referências Bibliográficas:

W. Duarte. "Delphi para Android e iOS: Desenvolvendo Aplicativos Móveis". Rio de Janeiro: Brasport, 2015.

J. Granatyr, G. Pauli, T. L. Lauxen, J. P. Barddal, F. Spak, F. Taffe. "Construindo Aplicativos Android no Delphi 10 Seattle", Clube Delphi, v. 166, 2016.

J. Granatyr, A. M. Paim. "ListView: Listagem de Dados dinâmica em aplicações mobile", Clube Delphi, v. 169, 2016.

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Personalização do ListView em aplicativos mobile

Aprenda a adicionar componentes em tempo de execução no ListView

Um dos principais componentes do FireMonkey para a construção de aplicativos móveis é o ListView, que tem a função de listar registros em tela em formato de grade assim como o DBGrid em aplicações desktop. Esse componente (o ListView) apresenta uma série de propriedades que permitem sua personalização, bem como a adição de rótulos, botões e imagens. Uma propriedade muito utilizada pelos desenvolvedores é a *ItemAppearance*, que disponibiliza uma série de valores para o posicionamento dos componentes em tela. Por exemplo, é possível configurar textos de cabeçalho agrupados, posicionamento de rótulos à direita e abaixo do texto principal, botões com ações para outros funcionamentos e pequenas imagens do tipo *thumbnail*.

Apesar de existir uma série de configurações bastante diversificadas, essa propriedade é estática e não permite uma personalização mais detalhada, como a configuração exata das posições X e Y dos rótulos ou a adição de novos recursos. Neste contexto, caso seja necessário personalizar ainda mais esse componente, é preciso entender alguns de seus eventos.

Baseado nisso, o objetivo do presente artigo é mostrar como construir um aplicativo no Delphi 10 Seattle utilizando o ListView e a adição dinâmica de novos componentes dentro de cada linha de dados. Será mostrado como carregar registros de um arquivo XML por meio de um *TClientDataSet* e do recurso de LiveBindings. O exemplo proposto lista informações sobre os lutadores do UFC em um ListView, e adiciona nele, em tempo de execução, um novo rótulo e três imagens representando dados armazenados no XML citado. Neste exemplo em particular, utilizaremos o número de *submissions*, *strikes* e *takedowns* de cada lutador e, ao invés de mostrar o valor numérico desses campos, os substituiremos por uma imagem de status que representará essas informações. As próximas seções mostram o desenvolvimento do aplicativo.

Fique por Dentro

O ListView é considerado o principal componente do FireMonkey para listagem de dados em aplicações mobile, porém ele não fornece muitas opções para a personalização avançada do componente em design time. Dessa forma, é necessário saber como manipular suas propriedades via código para obter efeitos avançados e que não estão disponíveis por meio da configuração de suas propriedades, como a adição de rótulos em tempo de execução ou a criação de ícones que representam o status de registros, por exemplo. Nesse contexto, é importante entender como criar classes específicas para a manipulação das informações de bases de dados, bem como relacioná-las às propriedades do ListView para obter efeitos interessantes e que melhoram o visual de um aplicativo mobile.

Nota

No UFC, a submissão ocorre quando um lutador coloca seu adversário em uma posição dolorosa em que ele não pode se mover ou escapar. Desse modo o lutador é obrigado a desistir ou a continuar resistindo a dor. O strike acontece quando várias partes do corpo são utilizadas para atacar ou pressionar o oponente, enquanto que takedown é fazer com que o lutador seja lançado ao chão.

Criando a interface gráfica e acessando os dados

O primeiro procedimento para a construção do exemplo que envolve os lutadores do UFC é a criação de uma nova aplicação multidispositivo no Delphi 10 Seattle (através do menu *File* > *New* > *Multi-Device Application*) com o layout em branco (*Blank Application*).

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

A **Figura 1** apresenta o formulário principal do aplicativo (*frmUFC*), no qual se encontra uma TToolBar (*tlbCabecalho*) e, dentro dela, um componente TLabel (*lblTitulo*) com as seguintes propriedades relativas à aparência:

- *Align* = "Contents"
- *Text* = "Lutadores do UFC"
- *TextSettings* > *Font* > *Size* = "20"
- *TextSettings* > *Font* > *Style* > *fsBold* = "True"
- *TextSettings* > *HorzAlign* = "Center"

No restante do formulário deve ser adicionado um TListView com a propriedade *Align* setada como *Client* para que o componente ocupe toda a área restante do formulário. Além disso, o ListView deve ter a sua propriedade *ItemAppearance* > *ItemAppearance* mudada para *ListItemRighDetail* para que seja possível exibir, além do rótulo, um texto de detalhe.

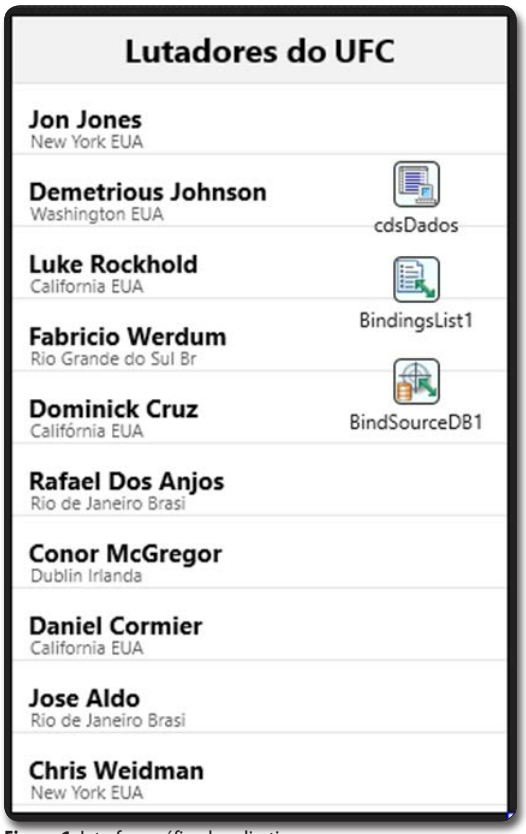

Figura 1. Interface gráfica do aplicativo.

Nota

A propriedade Align = "Contents" faz o alinhamento de um componente com base em todo o container em que ele está inserido. Por exemplo, caso exista um label à esquerda e se deseje adicionar um título no centro utilizando outro label, esse segundo deve utilizar a propriedade Contents para ser alinhado no centro de todo o container. Caso seja definida a propriedade Client, por exemplo, o label será alinhado somente no espaço vago entre os dois rótulos. Note que nesses casos, como queremos que o conteúdo do texto fique alinhado no centro horizontal do label, também é necessário que a propriedade TextSettings > HorzAlign seja configurada como Center.

Os dados que serão utilizados para preencher o ListView serão provenientes de um TClientDataSet, portanto esse componente também deve ser adicionado ao formulário (*cdsDados*). Não será realizada conexão à nenhuma base de dados pois os dados serão carregados do arquivo XML mostrado na **Listagem 1**.

Nota

Apesar de não ser tão comum, é possível desenvolver um aplicativo mobile utilizando arquivos XML como base de dados, bastando que, no momento do deploy da aplicação para o dispositivo físico, os arquivos sejam copiados para o device.

Listagem 1. Base de dados em formato XML.

```
01 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
02 <DATAPACKET Version="2.0">
03  <METADATA>
04   <FIELDS>
05   <FIELD attrname="Ranking" fieldtype="r8"/>
06   <FIELD attrname="Nome" fieldtype="string" WIDTH="20"/>
07   <FIELD attrname="Representa" fieldtype="string" WIDTH="20"/>
08   <FIELD attrname="Idade" fieldtype="r8"/>
09   <FIELD attrname="Submissions" fieldtype="r8"/>
10  <FIELD attrname="Striking" fieldtype="r8"/>
11  <FIELD attrname="Takedowns" fieldtype="r8"/>
12  </FIELDS>
13  <PARAMS DEFAULT_ORDER="1" PRIMARY_KEY="1" LCID="1033"/>
14 </METADATA>
15 <ROWDATA>
16  <ROW Ranking="1" Nome="Jon Jones" Representa="New York EUA"
      Idade="28" Submissions="5"  Striking="4" Takedowns="5" />
17  <ROW Ranking="2" Nome="Demetrious Johnson " Representa="Washington
      EUA" Idade="29" Submissions="4"  Striking="4" Takedowns="4" />
18  <ROW Ranking="3" Nome="Luke Rockhold" Representa="California EUA"
      Idade="31"  Submissions="4"  Striking="4" Takedowns="4" />
19  <ROW Ranking="4" Nome="Fabricio Werdum" Representa="Rio Grande do Sul
      Brasil"  Idade="38"  Submissions="5"  Striking="3" Takedowns="4" />
20  <ROW Ranking="5" Nome="Dominick Cruz" Representa="Califórnia EUA"
      Idade="31"  Submissions="3"  Striking="4" Takedowns="3" />
21  <ROW Ranking="6" Nome="Rafael Dos Anjos" Representa="Rio de Janeiro
      Brasil" Idade="31" Submissions="1"  Striking="2" Takedowns="5" />
22  <ROW Ranking="7" Nome="Conor McGregor" Representa="Dublin Irlanda"
      Idade="27"  Submissions="4"  Striking="4" Takedowns="4" />
23  <ROW Ranking="8" Nome="Daniel Cormier" Representa="California EUA"
      Idade="37"  Submissions="4"  Striking="2" Takedowns="4" />
24  <ROW Ranking="9" Nome="Jose Aldo" Representa="Rio de Janeiro Brasil"
      Idade="29"  Submissions="4"  Striking="3" Takedowns="3" />
25  <ROW Ranking="10" Nome="Chris Weidman" Representa="New York EUA"
      Idade="31" Submissions="5"  Striking="2" Takedowns="3" />
26  <ROW Ranking="11" Nome="Frankie Edgar" Representa="New Jersey EUA"
      Idade="34"  Submissions="5"  Striking="2" Takedowns="3" />
27  <ROW Ranking="12" Nome="TJ Dillashaw" Representa="California EUA"
      Idade="30"  Submissions="5"  Striking="2" Takedowns="2" />
28  <ROW Ranking="13" Nome="Joanna Jedrzejczyk" Representa=
      "Olsztyn Polônia" Idade="28"  Submissions="4"  Striking="2" Takedowns="4" />
29  <ROW Ranking="14" Nome="Miesha Tate" Representa="Washington EUA"
      Idade="29"  Submissions="3"  Striking="2" Takedowns="4" />
30 </ROWDATA>
31 </DATAPACKET>
```


DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

A primeira linha da **Listagem 1** é responsável pela declaração do arquivo e é composta da versão da linguagem e da codificação utilizada; além disso ela indica se o arquivo referencia um DTD (Definição do tipo de documento) ou uma entidade externa. A linha 2 apresenta a versão do pacote de dados, e na linha 3 o campo *METADATA* indica o cabeçalho do arquivo. Entre as linhas 4 e 12 são definidos os campos que serão utilizados para este exemplo, que são: ranking do lutador, nome, cidade e país que o lutador representa, idade, o número de submissões, strikes e takedows.

Na linha 13, a tag *PARAMS* realiza o controle das alterações e modificações, e entre as linhas 17 e 29 os lutadores são efetivamente cadastrados nessa base de dados. Por fim, a linha 30 representa a finalização da leitura de dados e a linha 31 finaliza o arquivo. É importante salientar que esse arquivo está no formato DataPacket, que é o padrão de arquivo que pode ser lido pelos TClientDataSets.

Clicando com o botão direito no ClientDataSet (*cdsDados*) e escolhendo a opção "*Load from Mybase Table*", uma janela será aberta, na qual o arquivo XML definido na **Listagem 1** deve ser selecionado para que os dados possam ser carregados para esse componente.

Depois, afim de exibir esses dados no ListView, é preciso efetuar a ligação entre ele e o ClientDataSet por meio do recurso de LiveBindings. Portanto, acesse o menu *View* e selecione a opção *LiveBindings Designer* para que uma tela similar a da **Figura 2** seja apresentada. É possível observar que o campo *Nome* foi associado à propriedade *Item.Text* enquanto que o campo *Representa* foi vinculado ao *Item.Detail*. Para realizar essa associação basta arrastar o campo *Nome* para *Item.Text* e o campo *Representa* para *Item.Detail*.

É ainda possível observar na **Figura 1** que dois novos componentes foram adicionados ao formulário: o TBindingsList e o TBindSourceDB. Esses dois componentes são responsáveis pelo gerenciamento das vinculações e ambos são acrescentados ao formulário automaticamente pelo RAD Studio.

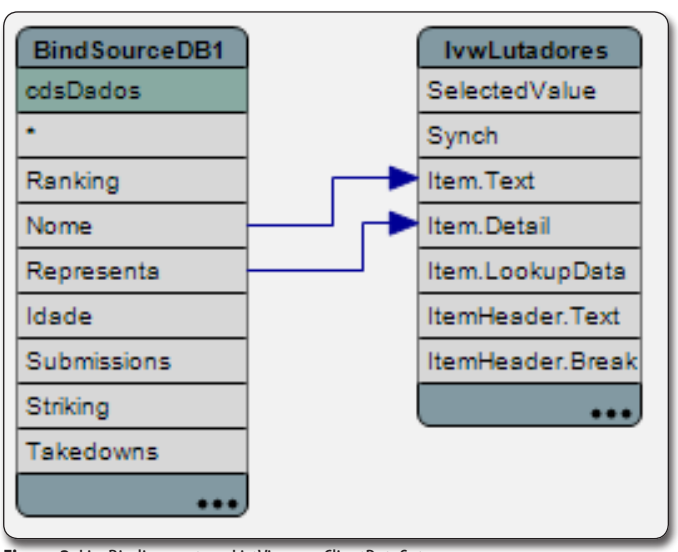

Figura 2. LiveBindings entre o ListView e o ClientDataSet

Nota

O Mybase tem a capacidade de ler conjuntos de dados a partir de arquivos baseados no padrão XML (Extensible Markup Language). O XML é uma linguagem de marcação utilizada para descrever dados estruturados, e o suporte a esses arquivos consiste em converter os pacotes de dados para a propriedade Data do TClientDataSet.

Para que o ListView possa apresentar o mesmo estilo visual da **Figura 1** são necessárias algumas configurações nesse componente:

1. A primeira é ocultar a seta que representa o acessório de mais informações por meio da seguinte propriedade:
   a. ItemAppearanceObjects > ItemObjects > Acessory > Visible = "False".
2. Para que o texto principal do ListView (nome dos lutadores, indicado pelo *Item.Text*) tenha aparência similar à apresentada na **Figura 1**, deve-se mudar as seguintes propriedades:
   a. ItemAppearanceObjects > ItemObjects > Text > Font > Size = "15"
   b. ItemAppearanceObjects > ItemObjects > Text > Font > Style = "fsBold".

Esse procedimento altera o tamanho da fonte e aplica o negrito para o texto, respectivamente.

3. Por fim, as seguintes propriedades devem ser alteradas para que o texto de detalhe do ListView (campo *Representa*, relacionado ao *Item.Detail*) fique semelhante à **Figura 1**:
   a. ItemAppearanceObjects > ItemObjects > Detail > Align = "Leading";
   b. ItemAppearanceObjects > ItemObjects > Detail > TextAlign = "Leading";
   c. ItemAppearanceObjects > ItemObjects > Detail > TextVertAlign = "Trailing";
   d. ItemAppearanceObjects > ItemObjects > Detail > PlaceOffset > X = 1;
   e. ItemAppearanceObjects > ItemObjects > Detail > PlaceOffset > Y = -1.

As três primeiras opções são relativas ao alinhamento do texto, enquanto que as duas últimas fazem com que o texto seja recuado e posicionado abaixo do texto principal. Por meio desses ajustes, o aplicativo já está funcional e com os dados sendo apresentados no ListView.

Nota

A propriedade ItemAppearance é responsável por definir a aparência gráfica do item, por exemplo imagem, legenda ou botão. Por outro lado, a propriedade ItemAppearanceObjects tem objetivo de gerenciar a personalização dos itens de exibição na lista.

Adicionando textos dinamicamente no ListView

A configuração da disposição dos componentes do ListView deve seguir conforme as opções disponíveis na propriedade *ItemAppearance*, a qual já fornece maneiras pré-determinadas para a adição

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

de dados nesse componente. Para adicionar novos componentes no ListView, deve-se criá-los dinamicamente, em tempo de execução, e posicioná-los manualmente. Como exemplo, a **Figura 3** mostra a adição da idade ao lado do nome do lutador enquanto que a **Listagem 2** apresenta a codificação correspondente para isso.

Figura 3. Adição da idade do lutador no ListView

Listagem 2. Adição de texto em tempo de execução.

```
01 procedure TfrmUFC.lvwLutadoresUpdateObjects(const Sender: TObject;
     const AItem: TListViewItem);
02 var
03   TextoIdade: TListItemText;
04 begin
05   TextoIdade := TListItemText.Create(AItem);
06   TextoIdade.Name := 'Idade';
07   TextoIdade.Align := TListItemAlign.Trailing;
08   TextoIdade.VertAlign := TListItemAlign.Center;
09   TextoIdade.TextAlign := TTextAlign.Center;
10  TextoIdade.PlaceOffset.X := -80;
11  TextoIdade.PlaceOffset.Y := 0;
12  TextoIdade.Font.Size := 13;
13  TextoIdade.Width := 35;
14  TextoIdade.Height := 18;
15 end;
```

Foi programado o evento *UpdateObjects* do ListView, o qual é executado para cada registro existente na base de dados. Na linha 3 é declarada a variável *TextoIdade* do tipo *TListItemText*, que representa um texto a ser adicionado como item do ListView. Na linha 5 esse objeto é instanciado por meio do construtor da classe, que recebe como parâmetro o item/registro atual do ListView que é passado por parâmetro para o procedimento. Dessa forma, cada texto criado terá como pai o item corrente do ListView, ou seja, cada um dos lutadores.

Na linha 6 é setado um nome para o componente, enquanto que nas linhas 7, 8 e 9 são configurados os alinhamentos horizontal, vertical e do texto. Na linha 10 é definido o posicionamento horizontal (X) do componente, o qual recebe um valor negativo (-80) para que fique posicionado mais para a esquerda e perto do nome do lutador — o valor zero coloca o componente exatamente na margem direita, então usamos números negativos para afastar o elemento da margem. Similarmente, na linha 11 é definido o alinhamento vertical (Y) do componente — o valor zero centraliza o componente, valores positivos o colocam para cima e valores negativos para baixo. Por fim, entre as linhas 12 e 14 são configurados o tamanho da fonte e a largura e altura do componente.

Como está sendo utilizado Live Bindings para vincular o ListView com o ClientDataSet, para que o texto seja exibido é preciso fazer a vinculação via código entre o campo idade criado dinamicamente no ListView e o campo idade do ClientDataSet. Para isso, a **Listagem 3** apresenta o código que realiza essa função, o qual deve ser programado no evento *LiveBindings > LinkFillControlToField1 > OnFilledListItem* do ListView *lvwLutadores*.

Nota

É importante salientar que no código da **Listagem 2** as linhas 7, 8 e 9 tratam do alinhamento do texto que está dentro do componente. Por outro lado, as linhas 10 e 11 dizem respeito ao alinhamento do componente em si.

Listagem 3. Adição de rótulo em tempo de execução.

```
01 procedure TfrmUFC.LinkFillControlToField1FilledListItem(Sender: TObject;
     const AEditor: IBindListEditorItem);
02 var
03   Item: TListViewItem;
04   TextoIdade: TListItemText;
05   TextField: TField;
06 begin
07   Item := lvwLutadores.Items[AEditor.CurrentIndex];
08   TextoIdade := Item.Objects.FindObject('Idade') as TListItemText;
09   TextField := BindSourceDB1.DataSet.FindField('Idade');
10  TextoIdade.Text := TextField.AsString;
11 end;
```

Na linha 3 é declarada a variável para representar o item atual do ListView (do tipo *TListViewItem*), enquanto que nas linhas 4 e 5 são declaradas variáveis para representar o texto de idade e o campo (dos tipos *TListItemText* e *TField*, respectivamente). Na linha 7 a variável *Item* recebe o item corrente do ListView, e na linha 8 é utilizado o método *FindObject* para encontrar o texto que está dentro desse item. Para encontrá-lo, é passado como parâmetro a string *Idade*, que foi definida na linha 6 da **Listagem 2**.

Similarmente, na linha 9 é utilizado o método *FindField*, herdado de TDataSet, para localizar o campo *idade* no ClientDataSet, por meio do *BindSourceDB1*. Finalmente, na linha 10 é feita a vinculação entre os dois campos, ou seja, para o texto da variável *TextoIdade* é atribuído o texto do campo por meio do método *AsString*.

Essa codificação equivale à ligação visual dos componentes apresentada na **Figura 2**. Código semelhante a esse pode ser utilizado em outras situações, nas quais é necessário vincular campos ou propriedades em tempo de execução, normalmente quando o recurso visual do LiveBindings não está disponível.

Com isso, o aplicativo já pode ser executado e terá o efeito parecido com o mostrado na **Figura 3**, com a idade do lutador sendo exibida ao lado do seu nome. Esse recurso de adicionar novos componentes ou dados dentro de um ListView é particularmente útil quando uma determinada tabela possui muitos campos importantes e vários deles precisam ser mostrados.

Nota

No Delphi 10 Seattle é possível instalar a visualização MultiDetail, que será exibida na propriedade ItemAppearance do ListView. Essa propriedade permite que mais três campos sejam adicionados, porém ainda assim não é possível definir um posicionamento muito personalizado. Consulte a seção **Links** para mais informações.

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Adicionando ícones personalizados no ListView

Para complementar nosso exemplo, vamos agora codificar uma classe que fará o desenho de três ícones a serem exibidos ao lado de cada lutador conforme o mostrado na **Figura 4**. Esses ícones serão dinamicamente desenhados de acordo com os valores dos campos *submissions*, *striking* e *takedowns*. Como pode ser observado nos valores desses campos no arquivo XML, eles estão na faixa de 1 até 5, portanto teremos cinco imagens que representarão esses números. Em suma, ao invés de mostrar no ListView os valores numéricos, mostraremos ícones que os representarão. Observe na **Tabela 1** os ícones e seus valores numéricos correspondentes.

No contexto do UFC, quanto maiores esses números, melhor é o desempenho do lutador. Na **Figura 4** é possível observar como o nosso ListView ficará de acordo com a base de dados.

O código da **Listagem 4** apresenta a definição da classe que será utilizada para construir os ícones e para representar um ícone dentro de um item do ListView. Esse código deve ser digitado na seção *interface* do formulário, e a unit *Generics.Collection* precisa ser adicionada na seção *uses* da janela principal.

Valor	Ícone
1	
2	
3	
4	
5	

Tabela 1. Ícones utilizados e seus respectivos valores numéricos

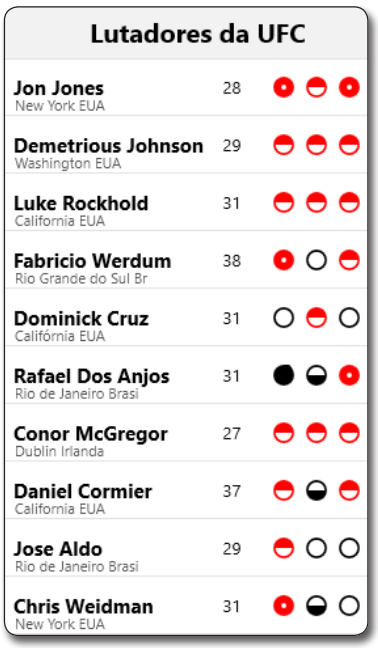

Figura 4. Adição de ícones personalizados no ListView

Listagem 4. Criação da classe TListItemRatingIcon.

```
01 type
02  TListItemRatingIcon = class(TListItemDrawable)
03  private
04    FRating: Integer;
05    FRatingPaths: TObjectList<TPathData>;
06  public
07    constructor Create(const AOwner: TListItem); override;
08    destructor Destroy; override;
09    procedure SetRating(Value: Integer);
10   procedure Render(const Canvas: TCanvas; const DrawItemIndex: Integer;
      const DrawStates: TListItemDrawStates; const SubPassNo: Integer = 0 );
      override;
11 end;
```

Na linha 2 é definida a classe que representa o *rating* do lutador com base em um dos três campos mencionados anteriormente. Essa classe herda de *TListItemDrawable* e é responsável pela implementação da interface de objetos persistentes. A variável *FRating* na linha 4 armazena o valor obtido de um dos campos (*submissions*, *striking* e *takedows*).

Na linha 5 é utilizada uma lista porque cada lutador armazenará todas as cinco imagens, sendo que a que será exibida no ListView será a que estiver de acordo com seu desempenho. Nessa lista serão armazenados objetos TPathData representando uma série de curvas e linhas conectadas, o que efetivamente cria os desenhos mostrados na **Figura 4**.

Por fim, entre as linhas 7 e 10 da **Listagem 4** estão declarados quatro métodos que serão posteriormente explanados em detalhes — para que eles sejam criados na seção *implementation* do formulário deve-se utilizar a combinação de teclas CTRL+SHIFT+C.

O próximo passo é criar o construtor da classe. Nele iremos popular a lista *FRatingPaths* com as cinco "imagens" apresentadas, todas do tipo TPathData. O TPathData pode ser inicializado definindo uma sequência de caracteres conforme o apresentado na **Listagem 5**.

Entre as linhas 6 e 11, é configurado o posicionamento, a altura e a largura do componente por meio de comandos semelhantes aos apresentados anteriormente para a configuração do rótulo da idade. Na linha 12 a variável *FRating* recebe o valor 1, ou seja, quando o objeto é criado ele possui o valor de *rating* correspondendo ao ícone preto com preenchimento. Essa inicialização é necessária somente para criar o objeto, pois posteriormente esse valor será alterado com base nos dados do arquivo XML.

Na linha 13 é criada uma lista de objetos do tipo TPathData que armazenará cada um dos cinco ícones a serem criados (note que o tamanho da lista foi definido na linha abaixo). O código presente entre as linhas 16 e 34 é responsável pela criação de cinco objetos TPathData, correspondentes a cada uma das imagens de rating.

A propriedade *Data*, setada em cada TPathData criado, representa uma string composta de caracteres que indicam os tipos e as coordenadas das linhas e curvas (veja a linha 17). O valor *M3002,0*, por exemplo, indica o ponto inicial na tela da forma a ser criada. O caractere *C* indica que será criada uma curva, definida por quatro pontos, e o caractere *Z*, por sua vez, é um parâmetro que

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

indica o final do caminho no PathData. Em suma, para que uma curva seja desenhada é passado o ponto inicial, e em seguida são definidos os dois pontos de controle e o valor com a posição final na tela. Com isso, entre as linhas 16 e 18 é desenhada a primeira bolinha, entre as linhas 20 e 22 a segunda e assim sucessivamente para os cinco ícones.

Listagem 5. Criação dos ícones no construtor da classe.

```
1 constructor TListItemRatingIcon.Create(const AOwner: TListItem);
2 var
3   PathData: TPathData;
4 begin
5   inherited;
6   Align := TListItemAlign.Trailing;
7   VertAlign := TListItemAlign.Center;
8   PlaceOffset.Y := 0;
10  Width := 16;
11  Height := 16;
12  FRating := 1;
13  FRatingPaths := TObjectList<TPathData>.Create;
14  FRatingPaths.Capacity := 5;
15
16  PathData := TPathData.Create;
17 PathData.Data := 'M3002,0 C3107,0 4003,896 4003,2002 C4003,3107 3107,4003
2002,4003 C896,4003 0,3107 0,2002 C0,896 896,0 2002,0 Z';
18  FRatingPaths.Add(PathData);
19
20  PathData := TPathData.Create;
21  PathData.Data := 'M3000,998 C4106,998 5002,1894 5002,3000 C5002,4106
4106,5002 3000,5002 C1894,5002 998,4106 998,3000 C998,1894 1894,998
3000,998 Z M3000,1448 C3857,1448 4552,2143 4552,3000 L1448,3000 C1448,2143
2143,1448 3000,1448 Z';
22  FRatingPaths.Add(PathData);
23
24  PathData := TPathData.Create;
25  PathData.Data := 'M2002,0 C3107,0 4003,896 4003,2002 C4003,3107 3107,4003
2002,4003 C896,4003 0,3107 0,2002 C0,896 896,0 2002,0 Z M2002,450 C2859,450
3554,1145 3554,2002 C3554,2859 2859,3554 2002,3554 C1145,3554 450,2859
450,2002 C450,1145 1145,450 2002,450 Z';
26  FRatingPaths.Add(PathData);
27
28  PathData := TPathData.Create;
29  PathData.Data := 'M2002,0 C3107,0 4003,896 4003,2002 C4003,3107
3107,4003 2002,4003 C896,4003 0,3107 0,2002 C0,896 896,0 2002,0 Z M3554,2002
C3554,2859 2859,3554 2002,3554 C1145,3554 450,2859 450,2002 L3554,2002 Z';
30  FRatingPaths.Add(PathData);
31
32  PathData := TPathData.Create;
33  PathData.Data := 'M2002,0 C3107,0 4003,896 4003,2002 C4003,3107
3107,4003 2002,4003 C896,4003 0,3107 0,2002 C0,896 896,0 2002,0 Z M2002,1513
C2272,1513 2490,1732 2490,2002 C2490,2272 2272,2490 2002,2490 C1732,2490
1513,2272 1513,2002 C1513,1732 1732,1513 2002,1513 Z';
34  FRatingPaths.Add(PathData);
35 end;
```

Ao final, cada uma das formas desenhadas é adicionada à lista *FRatingPaths* por meio do método *add*, conforme mostram as linhas 18, 22, 26, 30 e 34. É importante salientar mais uma vez que cada objeto dessa classe será composto por uma lista dessas cinco imagens, e o ícone a ser exibido no ListView será de acordo com um dos três campos mencionados anteriormente.

A **Listagem 6** apresenta o método destrutor, que somente libera a memória do objeto *FRatingPaths* liberando também todos os elementos que estão dentro da lista.

Nota

A classe TPathData faz parte do namespace FMX.Graphics do FireMonkey e representa uma série de curvas e linhas conectadas para o desenho de formas gráficas. Por exemplo, alterar o código da linha 17 da **Listagem 5** para PathData.Data := 'M3002,0 C200,0 4003,896 4003,2002 C400,3107 3107,4003 2002,4003 C900,4003 0,3107 0,2002 C0,896 896,0 2002,0 Z' irá gerar uma imagem com as curvas diferentes pois o código com a string C, que representa as curvas, foi alterado conforme o mostrado em negrito. O FireMonkey também disponibiliza algumas formas prontas que podem ser utilizadas. Consulte a seção **Links** para maiores informações e exemplos sobre o TPathData.

Listagem 6. Destrutor da classe.

```
1 destructor TListItemRatingIcon.Destroy;
2 begin
3   FRatingPaths.Free;
4   inherited;
5 end;
```

A **Listagem 7**, por sua vez, apresenta o método responsável por definir o valor do rating, que nada mais é do que a quantidade de *submissions*, de *strikes* ou de *takedowns* que está definida no arquivo XML. Na linha 3 é checado se o valor passado como parâmetro é diferente do valor atual, e, caso verdadeiro o novo valor é atribuído na linha 5. O método *Invalidate* da linha 6 aplica alterações nas listas de exibição quando ocorrem alterações de layout, ou seja, se nenhuma atualização está em curso esse método invoca o método *repaint* para redesenhar a lista de exibição.

Listagem 7. Método para setar o valor do Rating do lutador.

```
1 procedure TListItemRatingIcon.SetRating(Value: Integer);
2 begin
3   if FRating <> Value then
4   begin
5   FRating := Value;
6   Invalidate;
7   end;
8 end;
```

O procedimento mostrado na **Listagem 8** é responsável por renderizar as formas gráficas na tela. Esse processo ocorre quando o layout é modificado ou alterado por meio da alteração na posição do visor ou no tamanho do conteúdo. Na linha 10 é que efetivamente definimos qual das imagens será exibida, buscando dentro da lista *FRatingPaths* o índice que corresponde à respectiva imagem. Essa lista contém as cinco imagens, começando no índice 0 e indo até o índice 4 (por isso é que comando *FRating – 1* é utilizado).

Para valores de 1 a 3 a forma terá a cor preta de acordo com o comando *case* mostrado nas linhas 13 e 14. Por outro lado, caso esse valor esteja entre 4 e 5 a cor da forma será vermelha (linhas 15 e 16). Com isso podemos definir cores de acordo com o status dos registros. Neste exemplo o vermelho representa bons resultados enquanto que o preto indica resultados medianos ou ruins.

O método *PathData.FitToRect*, na linha 19, é responsável por desenhar o retângulo que irá receber os círculos que correspondem ao *FRating*. Por sua vez, o método *Canvas.FillPath* (linha 20) é

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

responsável por desenhar na tela, sendo que o primeiro parâmetro é responsável por receber a forma/ícone que será desenhado enquanto que o segundo especifica qual deve ser a transparência no preenchimento da cor. O valor *1.0* na linha 20 indica que a imagem será desenhada em sua totalidade de cores, enquanto que valores mais baixos tornarão a cor da imagem mais fraca.

Listagem 8. Renderização dos componentes.

```
1 procedure TListItemRatingIcon.Render(const Canvas: TCanvas;
2  const DrawItemIndex: Integer; const DrawStates: TListItemDrawStates;
3  const SubPassNo: Integer);
4 var
5   PathData: TPathData;
6 begin
7   if SubPassNo <> 0 then
8     Exit;
9
10  PathData := FRatingPaths.Items[FRating - 1];
11
12  case FRating of
13    1, 2, 3:
14      Canvas.Fill.Color := TAlphaColors.Black;
15    4, 5:
16      Canvas.Fill.Color := TAlphaColors.Red;
17  end;
18
19  PathData.FitToRect(LocalRect);
20  Canvas.FillPath(PathData, 1.0);
21 end;
```

Estando finalizada a classe que irá desenhar as imagens dentro do ListView, o próximo passo é criar três objetos do tipo *TListItemRatingIcon* dentro de cada item do ListView. Para isso, são utilizados praticamente os mesmos procedimentos executados para adicionar o rótulo ao lado do nome do lutador, com a diferença de que será criado um objeto dessa nova classe.

O código da **Listagem 9** é um complemento da **Listagem 2** e também deve ser adicionado no evento *UpdateObjects* do ListView. Na linha 4 é declarada uma nova variável, chamada *RatingIcon*, do tipo *TListItemRatingIcon*, e que, conforme explanado anteriormente, representará os ícones referentes a cada lutador. Na linha 7 é utilizado o método *FindObject* para verificar se o objeto já existe em cada uma das linhas do ListView, pesquisando pelo campo *Submissions*. Caso o componente ainda não esteja definido (linha 8), na linha 10 o novo objeto é instanciado e o código do construtor apresentado na **Listagem 5** é executado para que as formas gráficas sejam criadas para cada um dos registros. Na linha 11 é definido o nome do novo componente e na linha 12 é setado o seu posicionamento. O mesmo código se repete entre as linhas 14 e 29, com a diferença de que são criados os ícones para os campos *striking* e *takedowns*, variando também o seu posicionamento na propriedade *PlaceOffset.X*.

Por fim, o último passo para que possamos visualizar os ícones ao lado da idade de cada lutador é realizar a vinculação de LiveBindings via código, assim como foi feito na **Listagem 3** para a adição do rótulo. Para isso, na linha 5 é declarada a variável *RatingField* (do tipo TField) e na linha 6, um objeto do tipo TListItemRatingIcon, chamado *RatingIcon*. Depois do *begin* da linha 7 existe um comentário que indica que o código deve ser acrescentado logo abaixo da última linha da **Listagem 3**.

Listagem 9. Adição de objetos do tipo TListItemRatingIcon em tempo de execução.

```
1 procedure TfrmUFC.lvwLutadoresUpdateObjects(const Sender: TObject;
   const AItem: TListViewItem);
2 var
3   TextoIdade: TListItemText;
4   RatingIcon: TListItemRatingIcon;
5 begin
6   //código omitido referente à Listagem 2
7   RatingIcon := AItem.Objects.FindObject('Submissions') as TListItemRatingIcon;
8   if RatingIcon = nil then
9   begin
10    RatingIcon := TListItemRatingIcon.Create(AItem);
11    RatingIcon.Name := 'Submissions';
12    RatingIcon.PlaceOffset.X := -50;
13  end;
14  RatingIcon := AItem.Objects.FindObject('Striking') as TListItemRatingIcon;
15  if RatingIcon = nil then
16  begin
17    RatingIcon := TListItemRatingIcon.Create(AItem);
18    RatingIcon.Name := 'Striking';
19    RatingIcon.PlaceOffset.X := -25;
20  end;
21
22  RatingIcon := AItem.Objects.FindObject('Takedowns') as TListItemRatingIcon;
23  if RatingIcon = nil then
24  begin
25    RatingIcon := TListItemRatingIcon.Create(AItem);
26    RatingIcon.Name := 'Takedowns';
27    RatingIcon.PlaceOffset.X := 0;
29  end;
29 end;
```

Na linha 9 é feita a localização do componente *Submissions* que foi criado no passo anterior, e, logo após, na linha 10, o *RatingField* recebe o campo que efetivamente está cadastrado no arquivo XML.

Conforme mencionado anteriormente, esse processo é necessário para fazer a vinculação em tempo de execução utilizando o recurso de LiveBindings. Na linha 11 é verificado se as duas variáveis são diferentes de nulo, para que então, na linha 12, seja setado o *rating* do lutador de acordo com o que está armazenado na base de dados, sendo invocado o método *SetRating* criado na **Listagem 7**. É importante enfatizar que é esse método é o que efetivamente fará a alteração da imagem de acordo com os valores que estão no arquivo XML. Entre as linhas 14 e 22 o mesmo processo é repetido, com a diferença de que é feita a vinculação com os campos *striking* e *takedowns*.

Para exemplificar, considere o primeiro lutador da base de dados (Jon Jones), que tem, respectivamente, os seguintes valores para os campos *submissions*, *striking* e *takedowns*: 5, 4 e 5. A cada execução do código da **Listagem 10**, o objeto *RatingIcon* (do tipo TListItemRatingIcon) irá receber cada um desses valores e usá-los para a escolha da cor e do tipo de figura gráfica que será exibida no ListView.

Neste ponto, caso o aplicativo seja executado será exibido o mesmo efeito mostrado na **Figura 4**, ou seja, a idade do lutador

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

juntamente com os ícones que representam os campos *submissions*, *striking* e *takedowns*. Dessa forma, construímos uma classe específica para mostrar o desempenho dos lutadores de forma visual, o que facilita bastante a visualização dos dados em aplicativos mobile. Esse recurso é bastante interessante e pode ser personalizado de acordo com a necessidade do sistema, podendo ser utilizado, por exemplo, para indicadores de desempenho em softwares comerciais.

Listagem 10. LiveBindings entre o ClientDataSet e o ListItemRatingIcon.

```
1 procedure TfrmUFC.LinkFillControlToField1FilledListItem
  (Sender: TObject; const AEditor: IBindListEditorItem);
2 var
3  Item: TListViewItem;
4  TextoIdade: TListItemText;
5  RatingField, TextField: TField;
6  RatingIcon: TListItemRatingIcon;
7 begin
8  //código omitido referente à Listagem 3
9  RatingIcon := Item.Objects.FindObject('Submissions') as TListItemRatingIcon;
10  RatingField := BindSourceDB1.DataSet.FindField('Submissions');
11  if (RatingField <> nil) and (RatingIcon <> nil) then
12   RatingIcon.SetRating(RatingField.AsInteger);
13
14  RatingIcon := Item.Objects.FindObject('Striking') as TListItemRatingIcon;
15  RatingField := BindSourceDB1.DataSet.FindField('Striking');
16  if (RatingField <> nil) and (RatingIcon <> nil) then
17   RatingIcon.SetRating(RatingField.AsInteger);
18
19  RatingIcon := Item.Objects.FindObject('Takedowns') as TListItemRatingIcon;
20  RatingField := BindSourceDB1.DataSet.FindField('Takedowns');
21  if (RatingField <> nil) and (RatingIcon <> nil) then
22   RatingIcon.SetRating(RatingField.AsInteger);
23 end;
```

Outro ponto importante é que aplicativos móveis necessitam ter sua interface atraente e agradável para os usuários, e a utilização desses ícones pode também melhorar o visual de uma aplicação na qual existem muitos campos numéricos de difícil visualização, fazendo com que a aplicação seja melhor aceita no mercado.

Autor

Jones Granatyr
Doutorando em Informática bolsista CAPES e Mestre em Ciência da Computação bolsista CNPq, ambos na área de Inteligência Artificial. Trabalha em projetos de pesquisa relacionados à área de Inteligência Artificial, tais como Sistemas Especialistas, Mineração de Dados, Mineração de Textos e Sistemas Multiagente.

Autor

Aldo Marcelo Paim
Bacharel em Sistema de Informação, especialista em Banco de Dados, Administração e Desenvolvimento. É mestrando em Informática pela Pontifícia Universidade Católica do Paraná (PUCPR), atuando na área de mineração de dados e aprendizagem de máquina. Tem experiência de 10 anos com a linguagem Delphi.

Autor

Fabio Spak
Pós-graduando em Business Intelligence pela Universidade Positivo. Bacharel em Ciência da Computação pela Universidade do Contestado - UnC, no campus de Porto União/SC. Desenvolve trabalhos e artigos técnicos/científicos na área de Inteligência Artificial, especificamente na linha de pesquisa de Mineração de Dados e Sistemas de Recomendação.

Autor

Graziele Katlen
Técnica em Administração e aluna do curso de Sistemas de Informação da Unisep - União de Ensino do Sudoeste do Paraná, campus Francisco Beltrão/PR. Trabalha na área de desenvolvimento de software com as linguagens Delphi e Java. Atua também como Analista de Testes.

Autor

Fábio Alexandre Taffe
Mestre em Ciência da Computação pela Universidade Federal de Santa Catarina. Trabalho em projetos de pesquisa na área de Tecnologia da Informação pela União de Ensino do Sudoeste do Paraná – Unisep. É coordenador e professor do curso de Sistemas de Informação da Unisep de Francisco Beltrão – PR.

Links e Referências:

Aparências do ListView
http://docwiki.embarcadero.com/Libraries/Seattle/en/FMX.ListView.TAppearanceListView.ItemEditAppearanceName

Informações sobre o TPathData
http://docwiki.embarcadero.com/Libraries/Berlin/en/FMX.Graphics.TPathData.Data

Exemplos do funcionamento do TPathData
http://docwiki.embarcadero.com/images/Libraries/Berlin/e/d/d0/FMXTypesTPathData-CurveTo.PNG

Documentação da Embarcadero
http://docwiki.embarcadero.com

Effectively Using List Controls in Object Pascal Mobile Apps
http://delphi-insider.blogspot.com.br/2015/03/effectively-using-list-controls-in.html

UFC – Ultimate Fighting Championship – Oficial Web Site
http://www.ufc.com.br/

Mobile no Delphi: Mudando a visualização dos dados no ListView
http://www.devmedia.com.br/mobile-no-delphi-mudando-a-visualizacao-dos-dados-no--listview/37115

W. Duarte. "Delphi para Android e iOS: Desenvolvendo Aplicativos Móveis". Rio de Janeiro: Brasport, 2015.

J. Granatyr, G. Pauli, T. L. Lauxen, J. P. Barddal, F. Spak, F. Taffe. "Construindo Aplicativos Android no Delphi 10 Seattle", Clube Delphi, v. 166, 2016.

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Somos tão apaixonados por tecnologia que o nome da empresa

diz tudo.

Porta 80 é o melhor que a Internet pode oferecer para sua empresa.

Já completamos 8 anos e estamos a caminho dos 80, junto com nossos clientes.

Adoramos tecnologia.
Somos uma equipe composta de gente que entende e gosta do que faz,

assim como você.

Estrutura

100% NACIONAL. Servidores de primeira linha, links de alta capacidade.

Suporte diferenciado

Treinamos nossa equipe para fazer mais e melhor. Muito além do esperado.

Serviços

Oferecemos a tecnologia mais moderna, serviços diferenciados e antenados com as suas necessidades.

1-to-1

Conhecemos nossos clientes. Atendemos cada necessidade de forma única. Conheça!

Porta 80
WEB HOSTING

Hospedagem | Cloud Computing | Dedicados | VoIP | Ecommerce | Aplicações | Streaming | Email corporativo

porta80.com.br | comercial@porta80.com.br | twitter.com/porta80

SP 4063-8616 | RJ 4063-5092 | MG 4063-8120 | DF 4063-7486